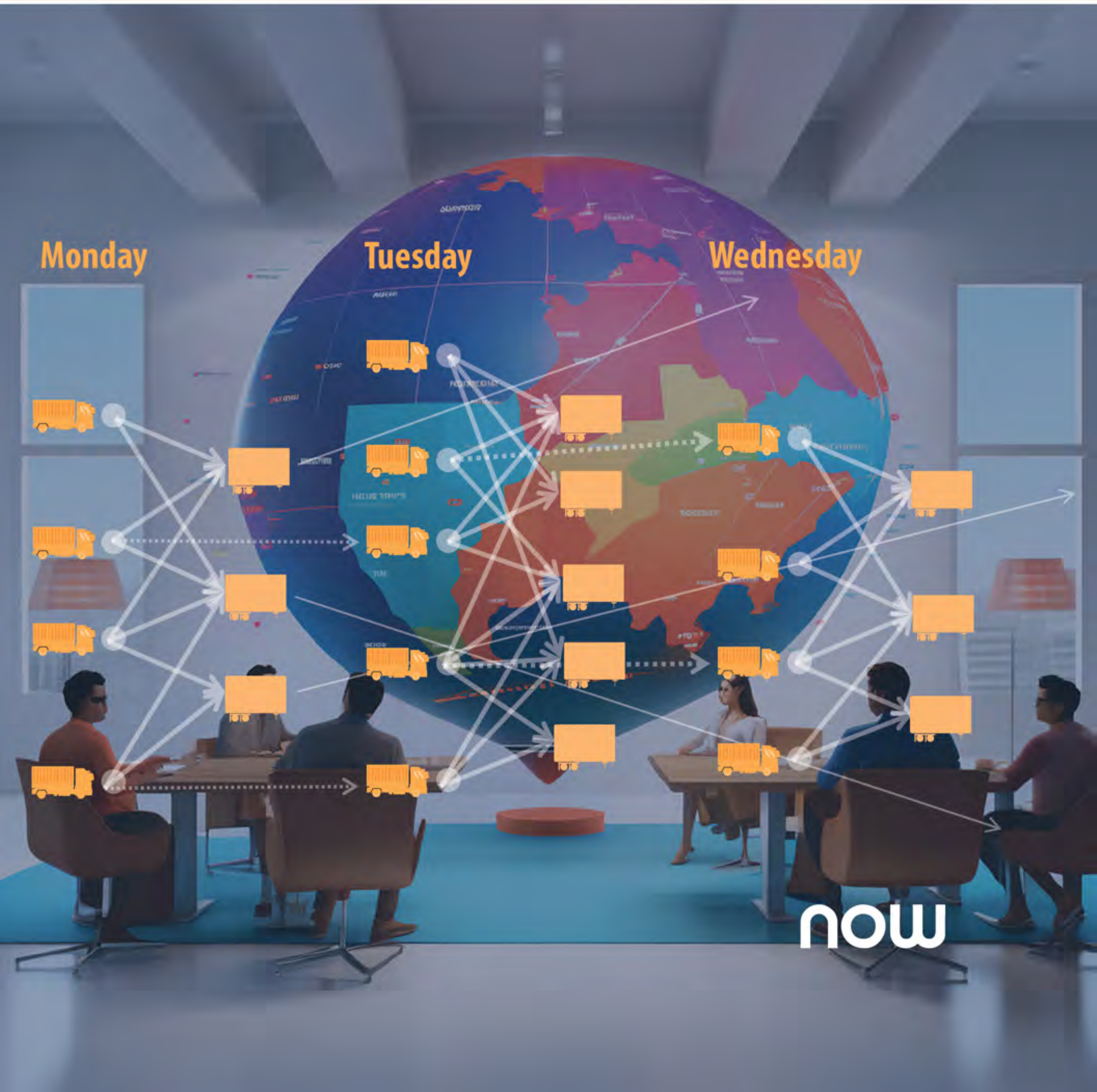
A Modern Approach to Teaching an Introduction to Optimization

Warren B. Powell



now

A Modern Approach to Teaching Introduction to Optimization

A Modern Approach to Teaching Introduction to Optimization

Warren B. Powell
Professor Emeritus, Princeton University
Chief Innovation Officer, Optimal Dynamics
Wbpowell328@gmail.com

now
the essence of knowledge
Boston — Delft

Published, sold and distributed by:
now Publishers Inc.
PO Box 1024
Hanover, MA 02339
United States
Tel. +1-781-985-4510
www.nowpublishers.com
sales@nowpublishers.com

Outside North America:
now Publishers Inc.
PO Box 179
2600 AD Delft
The Netherlands
Tel. +31-6-51115274

The preferred citation for this publication is

Warren B. Powell. *A Modern Approach to Teaching Introduction to Optimization*, pp. 1–116, 2023.

ISBN: 978-1-63828-321-8
© 2023 Warren B. Powell

Contents

Summary

"Optimization" is widely taught in departments such as operations research, industrial engineering, and sometimes applied math, as focusing on complex, multidimensional (and often very high-dimensional) problems that can be formulated as linear, nonlinear or integer programs. Introductory courses are often centered on linear programming, the simplex algorithm and duality theory.

"Optimization" should be the study of making good decisions, and should start with the simplest (but nontrivial) decisions that are familiar to every student. Linear programs solve a very tiny fraction of decision problems, even in areas such as business where linear programming is often taught. I will note that almost no-one in business without formal training in linear programming has even heard the term. More distressingly, only a fraction of undergraduates or masters students who take linear programming ever solve a linear program (even with a package). And no-one outside a tiny core of specialists has ever programmed the simplex algorithm.

This document outlines a new way of teaching optimization that starts with some basic machine learning problems that are very familiar today given the attention that "AI" has attracted. Few people recognize that machine learning is actually solving a stochastic optimization problem by assuming we are given a training dataset. I leverage this idea to introduce students to some simple sequential decision problems that involve decisions that everyone faces. We make these decisions using methods (policies) that range from simple, parameterized rules or functions that can be optimized exactly as we solve our machine learning problem.

I then present a series of topics that progress from simple problems that are familiar to everyone, eventually reaching topics such as linear, integer, and nonlinear programming. However, I minimize the attention given to the design of algorithms given that there are widely available packages (it will be the rare beginning student who ever transitions to an algorithmic designer). Instead, I focus on modeling and evaluating the solution which is typically done in the presence of uncertainty.

This book is aimed at faculty who are already teaching an introductory optimization course, or who have a background in optimization and are designing an optimization course. It is also useful to anyone with conventional training in optimization, since it will show you how to think about optimization differently. The presentation consists of a set of topics to guide the design of lectures, leaving considerable flexibility in terms of how much emphasis is placed on individual topics.

The presentation starts with sequential problems, since these are the simplest nontrivial decision problems that are most familiar to students. It then transitions to classical static optimization problems (linear, integer and nonlinear programming) but in each case we show how static optimization models can often be viewed as methods for making decisions in a sequential setting.

A Modern Approach to Teaching Introduction to Optimization

Warren B. Powell[1]

[1] *Professor Emeritus, Princeton University, Chief Innovation Officer, Optimal Dynamics; Wbpowell328@gmail.com*

Warren B. Powell (2023), "A Modern Approach to Teaching Introduction to Optimization", pp 1–116. DOI: 10.1561/9781638283218.

1

Introduction

Making decisions is a universal human activity, something we all have done since we are born. Making good decisions is how we perform better, whether we are running a business, managing a health or energy system, designing and controlling a supply chain, inventing new products and materials, or creating new drugs. Optimization, I claim, is the science of making the best decisions that we can.

The academic community has largely limited the scope of optimization to pet methods for different communities. Industrial engineering and operations research equate optimization with linear programming (as a starter), progressing into even more specialized fields such as nonlinear and integer programming. Dynamic programming is considered a very advanced topic, rarely taught at the undergraduate level. Faculty in engineering (mechanical, electrical, chemical) and economics focus on the field of optimal control, a close cousin of dynamic programming, typically using fairly advanced mathematics. Computer scientists will either study combinatorics (a close cousin of integer programming) or, more recently, reinforcement learning (a cousin of dynamic programming). All of these approaches to optimization are typically taught at a fairly advanced level.

I am going to suggest a new way to teach introductory optimization to undergraduates and masters that is far more useful and addresses the modeling and algorithmic challenges that arise in decision problems that occur in practice. To do this, I am going to take on two of the titans of optimization: George Dantzig and Richard Bellman. In this course, we will still teach linear programming, but it is demoted to a method that is used only for a small number of problems, and while I will illustrate the simplex algorithm using a network problem (where the steps of the simplex algorithm can be described visually), I do not drag students through the simplex algorithm for general linear programs.

By contrast, we will pay considerably more attention to sequential decision problems (also known as dynamic programs), but we will largely ignore Bellman's equation (or Hamilton-Jacobi equations). Also, we note that the origins of linear programming were in the context of sequential decision problems, and we will make the distinction between linear programming as a static problem, and as a policy for sequential decision problems. We do the same for integer and nonlinear programming.

My approach will be to start with the simplest nontrivial problems, and progress to more complex settings. We start with some basic machine learning problems (with linear and nonlinear models) that lay the foundation for modeling and solving sequential decision problems (SDPs). SDPs, which usually involve making relatively simple decisions over time, are problems that are pervasive in business, economics, engineering, the sciences, and even everyday life. SDPs include problems where the decision may be binary (which web page design to use, when to stop and sell an asset), discrete (choosing the best catalyst, drug, product, path, supplier, employee, . . .), and scalar continuous (finding the best price, dosage, temperature, budget). We make decisions using the three classes of policies that are widely used in practice (albeit in an ad hoc way), but show students how to properly model and tune these policies. And we do not use Bellman's equation other than a brief illustration in a shortest path problem. Bellman requires a level of sophistication for both modeling and algorithmic work that is not suitable for an introductory course, and it is only useful for a very narrow subset of problems.

In the process of using sequential decision problems, we are going to have to introduce the dimension of modeling sequential information processes. While this potentially opens a pandora's box into the complex arena of stochastic modeling, we avoid this by exclusively working with random samples, just as is done with machine learning where a training dataset is a sample of the random observations.

The course is aimed at undergraduates (or masters) using a minimal amount of calculus or linear algebra. We use very simple Monte Carlo simulation, but do not require a course in probability or statistics. There are many opportunities for a faculty member to adjust the examples and scope of the material to their own students. However, we urge faculty to resist the typical style of teaching this material which emphasizes the methods that have a strong theoretical foundation. This material can be taught at a very advanced level, but this course focuses on teaching students how to *think* about making decisions.

2

Audience

This book was originally written to help schools teaching introductory optimization courses adjust their pedagogical approach to a modern style that is much more useful and relevant to students. For this purpose, I discuss below the audience from three perspectives: what departments can use this approach, the students that I am targeting, and the faculty who might be interested in teaching this.

A fourth audience is people who have already taken a traditional course in (typically deterministic) optimization. Simply reading these notes will provide a different perspective that draws on the skills you have already learned.

2.1 Academic Departments

I think the style of this course can be used in any department that involves making quantifiable decisions. This includes engineering (all departments), the physical sciences (laboratories are full of sequential decision problems), the social sciences, economics, business schools, politics, and psychology.

"Optimization" (linear, nonlinear and integer programming) is traditionally taught in operations research, industrial engineering and applied math. In engineering there is much more emphasis on control theory (a form of sequential decision problem), while economics (along with advanced courses in OR and IE) will emphasize dynamic programming. Computer science for the past decade has taught reinforcement learning, a field that addresses "Markov decision processes" which is just a different form of control problem. These courses all tend to be taught with a moderate to high level of mathematical sophistication, ignoring the reality that virtually everyone needs to solve sequential decision problems.

I will note that none of these classical courses on optimization traditionally deal with what I call "optimal learning" problems, which are problems where the decision controls what information to collect to improve your understanding of a process so you can make better decisions in the future. Optimal learning problems arise in laboratory experimentation (either physical experiments or computer simulations) and a host of field settings (what is the best product to recommend, what is the best price to charge, what is the best medical treatment, what process to use to make a material, ...). I taught a course called Optimal Learning for 10 years at Princeton at the undergraduate level. The course was quite popular, and deals with problems that are much more familiar to our students than linear programs.

The course I am proposing covers both static problems (such as linear programs) as well as sequential decision problems (which includes dynamic programming, optimal control and reinforcement learning). Topics like linear programming can be ignored (for fields where these problems simply do not arise), covered very briefly (a single lecture), or extensively (some courses will spend 4–6 weeks just on linear programming).

The real novelty of our teaching style is how we approach sequential decision problems, which arise in virtually every problem domain, and yet are often ignored in introductory courses in optimization. Our approach to teaching this topic emphasizes practical solution methods that reflect what is used in practice but placed in a framework that formalizes the evaluation and tuning of policies.

Our point of departure from traditional approaches for teaching sequential decision problems is that we largely ignore methods based on solving Bellman's equation (known as Hamilton-Jacobi equations in the optimal control field). Methods based on HJB equations are mathematically elegant, but only apply to a very narrow set of problems, which is a reason why people routinely make decisions over time who have never even heard of HJB equations.

[Side note: I spent 20 years of my career, and wrote a popular 500-page book on approximate dynamic programming (also known as reinforcement learning) which is a field that focuses on methods for solving HJB equations approximately. My conclusion that this approach has limited practical value is based on many years of research.]

2.2 Students

The course is aimed at undergraduates or masters students with no prior training in optimization. The following skills will be useful:

- Students will need some calculus, but only at the level of understanding a derivative and gradient (which, to be honest, can be presented very quickly). When we do use derivatives/gradients, these can often be estimated numerically instead of using the analytical formulas stressed in introductory calculus courses.

- We use a very modest amount of linear algebra – much less than traditional courses in optimization. For example, there are perhaps two places where we use the concept of an inverse of a matrix. Students without any prior training in linear algebra could be taught the basic idea of a vector and matrix in a short tutorial session.

- We will occasionally use some very basic concepts from statistics, but we will do this in a way that does not require a prior course in statistics. For example, it is very easy to introduce a student to a mean and variance. In this course there is no need for familiarity with different probability distributions.

2.3 Faculty

I am assuming that the faculty teaching this course are already trained in the core fields of linear, nonlinear and integer programming if you wish to cover this material, but there are entire fields (such as computer science) where students are typically not introduced to linear programming. These topics are covered, but not nearly in the depth that the faculty member might remember from their own training. Times have changed, and there is no longer a need to teach, for example, algorithms for solving linear programs (packages will be used).

2.4 Students/Professionals with Prior Optimization Training

Reading (even skimming) these notes by someone who has already had a course in (deterministic) optimization should change how they think the solution of the optimization model should be evaluated. This means appreciating that in many (most? almost all?) settings, deterministic optimization models are actually policies (methods for making decisions) that need to be evaluated over time, under uncertainty. This then opens a door to improving the solution of their deterministic model in terms of real-world performance.

3

Course Outline

This section provides a sketch of the course. The material is divided into 11 topics, which are described in much more depth in Section 5. Here, I summarize each of the topics, focusing on the development of key concepts.

The course will transition from the simplest (but nontrivial) decision problems to more complex settings. We will start with basic machine learning problems partly because they are very familiar today, but also because they are well motivated and easy to understand. The machine learning problems will also lay the foundation for how we handle random observations in sequential decision problems which are of fundamental importance (since so many real problems are, in fact, sequential in nature).

The lectures are organized to follow a natural progression from simpler decisions (binary, discrete, continuous scalar) to more complex ones (continuous vectors, integer variables, nonlinear functions). Our emphasis is on formulating optimization models, including the critical (but historically overlooked) problem of understanding how to handle decision problems when they are made sequentially over time. We present algorithms when one or both of the following applies:

- An understanding of the algorithm can help students appreciate the behavior of the solution, even if they never implement an algorithm.

- Students may need to program the algorithm if they are to solve the problem (we limit these to relatively simpler algorithms).

This document presents the course as a series of “topics” that can be adapted to the interests of the faculty member, and the background and interests of the students. Most topics can be presented in 1 to 3 lectures, but some topics can be extended to as many as 6–8 lectures depending on the interests of the faculty member and the skills and background of the students.

Below I will list each topic and describe the key points being covered, emphasizing the transition from simpler to more complex problems. This is a sharp departure from introductory courses in “optimization” that turn out to be courses in “linear programming with extensions.”

- Topic 1 – Machine learning – We start with fitting a linear model to introduce the idea of solving a convex optimization problem exactly. Students should learn the minimum amount of data required to fit a linear model (for example, $n \geq p$) in addition to other conditions on the data. We then transition to nonlinear models and introduce gradient search and the issue of multiple optima (a major topic with neural networks that are so prominent today). Also note that with nonlinear estimation, we no longer require $n \geq p$, implying that we can fit, say, a neural network with 100 million parameters with a single datapoint.

  A particularly important piece of pedagogy in Topic 1 is the idea (often overlooked) that estimation problems are, in fact, stochastic optimization problems, where random variables are replaced with sampled observations. This is going to set the style for handling uncertainty which will run throughout our handling of sequential decision problems. We note that this style allows students to do “stochastic optimization” without any training in stochastic optimization, probability, or even a course in statistics.

- Topic 2 – Sequential decision problems – We next introduce the concept of a sequential decision problem, and then use some important and visible problems to illustrate how to model these. Most important is the concept of a *policy* which is a method (that is, a function) for making decisions, that is controlled by tunable parameters. This closely parallels fitting a model to data (as we do in Topic 1). We start with a basic example for selling an asset that uses historical data, and then transition to an inventory problem where we need to randomly generate observations. This is done using a very basic introduction to Monte Carlo sampling.

  The policies we introduce are both forms of policy function approximations (PFAs) which are widely used by individuals as well as corporations. PFAs help us create a natural bridge to estimating functions in machine learning, but students also learn how to set up an objective function for sequential decision problems. This skill will stay with us as we progress to more complex decision problems.

- Topic 3 – Adaptive optimization – In this topic I use the newsvendor problem to introduce the idea of using sampled information to compute a gradient. This is widely known as a stochastic gradient in the literature, but the gradient is based on a sample, which means we are taking the derivative of a deterministic function. This problem will require generating random variables dynamically rather than creating a sample in advance as we did in Topic 2. It is important to recognize that while the newsvendor problem is perhaps the most widely studied stochastic optimization problem, the algorithm is quite simple, and outside of generating random samples, all of the steps use deterministic methods.

- Topic 4 – Optimal learning – This is a topic where the optimization problem is making decisions of what to observe, such as how a patient responds to a drug, how many clicks a website attracts, and the market demand for a product at a particular (discretized) price (applications of this model are endless, and familiar to everyone). We use a policy called interval estimation (a form of upper confidence bounding) that is very popular with tech companies (e.g. Google and Facebook) for maximizing ad-clicks. Optimal learning will also play a role any time we need to do parameter tuning, which will turn out to be a common type of optimization problem.

Upper confidence bounding policies represent a form of "cost function approximation" (CFA) where the policy for making decisions has an imbedded optimization problem. For UCB policies, this optimization problem involves a simple sort, but it introduces the idea of a policy that involves solving an optimization problem to make a decision. This means we have an optimization problem which requires sorting a set of estimates within a larger optimization problem of tuning the parameter in the UCB policy.

- Topic 5 – Shortest path problems – Here we introduce our first nontrivial static, deterministic optimization problem which is also a very special form of linear program (but that comes later). This is the only time we use Bellman's equation in the course, although there are problems (in Topic 10) where we could draw on Bellman again.

  After presenting the model and algorithm for a deterministic, static shortest path problem, we then transition to show how this can be used in a dynamic setting, as would happen if we are modeling a path through a dynamic network. We show how to model this problem, and then show how to create a classic deterministic dynamic lookahead approximation (deterministic DLA). We show how to evaluate the shortest path problem as a policy, and how to parameterize it so that it works better in a stochastic, dynamic environment.

  We are going to copy this setting as we move into more general optimization problems. We will start by presenting a basic, static optimization problem (this could be a linear, nonlinear or integer program), and then show how it is often used as a policy in a dynamic setting. In my experience, the vast majority of "optimization problems" are actually policies used in a sequential problem setting, something that is typically overlooked in standard texts on optimization.

- Topic 6 – General concepts – We pause at this point to discuss two important dimensions of sequential decision problems:
    - Classes of policies – So far, we have illustrated four ways of making decisions, each of which come from the four classes of policies. These four classes cover every possible method that we might use to make decisions, including any method people are already using.
    - Evaluating policies – The biggest difference between people who make decisions in an ad hoc way versus someone with formal training is their understanding of the concept of a policy, and how to evaluate it. In this topic (typically a single lecture) we start by reviewing how we have evaluated policies in topics 2 – 5. We then list different ways of evaluating policies such as cumulative reward for online learning, and final reward for learning in a lab. We also differentiate between expected performance versus risk. While the academic literature deals with risk with a considerable amount of mathematical sophistication, we are going to show students how to model risk in a way that can be easily computed in a spreadsheet.
- Topic 7 – Linear programming – Here is where we introduce linear programming. This can be done in a single lecture (which I recommend for an introductory optimization course) or expanded given the time available, interests of the students, and the interests of the faculty member teaching the course.

  Our preferred style for an introductory course is to teach the idea of a linear program and then transition to the understanding that "algorithms exist" for solving it. An in-depth presentation of the simplex algorithm is simply not appropriate at this stage, since no-one is ever going to implement their own simplex algorithm. Modern implementations of the simplex algorithm use a variety of sophisticated strategies to improve performance; these strategies are never discussed in textbook treatments of the simplex algorithm, so it is not clear what a student is learning from these streamlined presentations. In addition, production implementations might combine strategies such as dual simplex or even interior point methods. This material is simply not appropriate for an introductory course.

This said, we illustrate the simplex algorithm graphically using a network problem which helps students understand, in a highly visual way, the concept of a basis, pivoting, and most important, dual variables. This can be done without any linear algebra, but we do have a section where we present the simplex algorithm (for a network problem) both graphically, and then algebraically. We leave it to the instructor to decide which presentation best suits their students.

We begin by presenting linear programming as the solution to a static problem, but we then transition to using linear programming as a policy for sequential decision problems. We suspect that most linear programming applications arise in the context of sequential decision problems (and this is certainly true of the original motivating applications used by George Dantzig). I think the recognition that linear programs are often used as policies for sequential decision problems is one of the great failures of the math programming community. Topic 8 illustrates how a deterministic linear program might be used in a sequential inventory problem.

- Topic 8 – Dynamic inventory problem – Here we are going to copy what we did for our dynamic shortest path problem but use the context of an energy storage problem in a highly dynamic setting with rolling forecasts (a topic that has been completely overlooked in the operations research literature). This requires solving a series of simple linear programs, even though the decision at a point in time is a scalar (we get the LP because we are optimizing over a planning horizon, which means our decision variable is now a vector). The lookahead LP with be parameterized to help mitigate the errors in the rolling forecasts, and we will show that this produces a much better result than using typical point forecasts. The challenge, as always, will be the tuning, a problem we first saw in the machine learning problem in Topic 1. We will suggest a strategy that is fairly easy to implement.

- Topic 9 – Integer programming – Here we introduce the idea of integer variables in the context of a facility location problem. As with our shortest path problem, we will start with a simple, static facility location problem. Then, Topic 10 shows how the static model can be used as a policy in a fully sequential problem.

  Optimization books tend to become drawn into the fairly sophisticated algorithms required to solve integer programs. However, since year 2000, commercial packages have conquered wide classes of even very large integer programs, although some care has to be used since there is a wide range of integer programming problems, and some still require specialized algorithms. The best packages (such as Gurobi and Cplex) can be dramatically better than free software that students can download over the internet. As with algorithms for linear programs, teaching algorithms for integer programs is pointless for an introductory course – these algorithms are quite sophisticated and no-one today would implement their own. However, it is important for students to be able to recognize which types of integer programming problems are likely to be solvable with a general purpose package.

- Topic 10 – Dynamic facility location – As we did with linear programming, we start by presenting a static integer programming problem using facility location, and then extend it here to a dynamic setting. We start by making the case that any facility location problem would have to be implemented in a stochastic environment. We separate the decision of where to locate facilities, which is made using forecasted demands, and the "real world" decisions of how to meet demands which are revealed after we make the decision to locate facilities.

  We then recognize that decisions to locate facilities are themselves decisions that are made over time, in the presence of the uncertainties about demands. We illustrate a few strategies for solving the dynamic facility location problem.

- Topic 11 – Nonlinear programming – We have already seen nonlinear programming in topic 1 when we fitted a nonlinear model, but here we are going to address this rich topic in more depth. As with linear and integer programming, we are going to present nonlinear programming in two stages: first as a static problem, and then as a policy in a fully sequential problem. Nonlinear programming is a rich topic that can be introduced in a single lecture but can span an entire course. It is up to the professor to decide how much time to spend on this topic given the interests of the students.

  We are going to introduce students to a quadratic programming problem that arises when optimizing investments over a portfolio. We will first introduce this problem in its classical formulation as a static problem, and then transition to solving it sequentially over time, treating it as a policy in a fully sequential decision problem (based on actual practice on Wall St.). This will be a sophisticated (but very real) extension of the asset selling problem we introduced in Topic 2.

4

Readings

Many of the topics are organized around sequential decision problems presented in

Warren Powell, *Sequential Decision Analytics and Modeling*, NOW Press, 2022 (available for free download from https://tinyurl.com/sdamodeling). Below I refer to this as "SDAM."

Readings from SDAM are indicated at the beginning of each topic (or subtopic).

Occasionally I refer to material in my graduate-level book:

Warren Powell, *Reinforcement Learning and Stochastic Optimization*, Wiley, 2022 (see https://tinyurl.com/RLandSO/ for an overview). Below I refer to this as "RLSO."

RLSO is not appropriate for an introductory course such as this, but I recommend that the instructor have a copy of the book.

There are blocks of material on mature topics like linear, integer, and nonlinear programming. I assume that any professor teaching a course in optimization will already have a favorite book they like to use for these topics. We encourage, for an introductory course like this, putting more emphasis on describing what these problems are and how they are used, with less emphasis on algorithms, especially when these are widely available in packages.

5

Lectures

In this section I sketch out a sequence of topics that steadily transition from relatively simple decision problems to more complex ones. There is considerable flexibility in terms of how much time is spent on each topic. For example, linear programming can easily be taught in a single lecture (basically defining what a linear program is), but it can also fill half a course. There are also topics on integer programming and nonlinear programming which can also be taught in a single lecture, but there are entire (graduate level) courses dedicated to each of these topics. In an introductory course, I think students should be introduced to these topics, but in a modest way.

Topic 0: Applications

It always helps to start an introductory course such as this with a series of applications. This will be very dependent on the department where the course is being taught. Below I give some illustrative applications that I used when I was teaching this material.

- Machine learning problems – These require optimizing parameters to make a model fit a training dataset.

- Management of physical resources – Physical resources might be:
    - People (hiring, firing, training, moving)
    - Equipment (trucks, drones, robots, medical equipment, . . .)
    - Facilities (building, leasing, closing, resizing)
    - Product (planning inventories for retail sale, or parts used in manufacturing)
- Management of financial resources – These include
    - Planning cash reserves
    - Making investments
    - Arranging different financial instruments (loans, insurance contracts, . . .)
    - Setting budgets
- Information acquisition and communication
    - Running experiments in a lab or the field
    - Running medical tests
    - Sending/sharing information about the status of a system
- Finding the best ways of making decisions
    - Choosing the best method for making decisions
    - Tuning parameters used by a method

These decision problems can come in two forms:

- Static problems – These are problems we solve once and then use the solution
- Sequential decision problems – These are decisions that are made repeatedly over time as new information is arriving.

Sequential decision problems are quite rich, and typically involve relatively simple decisions. However, the sequential nature, and in particular the flow of new information, can introduce significant complexities.

We are going to steer around these complexities and show how to model and solve problems that everyone encounters in their own personal activities, or any of a wide range of problems in business, engineering and the sciences.

The powerpoint slides I used in my first lecture can be downloaded by going to https://tinyurl.com/RLSOcourses/. Scroll down to the heading "Undergraduate/masters course in sequential decision analytics" and then scroll down to "Lecture 1" and download the slides. However, it is very important that these applications be chosen based on the interests of the students.

Topic 1: Machine Learning

One of the most visible (and accessible) optimization problems today arises in machine learning, where we have to find the best fit of a model to a training dataset (this is also known as supervised machine learning).

Below we describe the optimization problems and solution methods that arise when we are fitting linear models and nonlinear models. Each setting will allow us to illustrate different optimization strategies, from finding an optimal solution analytically with linear models to using a derivative-based search algorithm for nonlinear models. We will also learn some properties of optimal solutions along with the necessary conditions for optimality in each setting.

1.1 Linear Models

We are going to start by assuming we have a basic training dataset that we can write as

$$(x^1, y^1),\ (x^2, y^2), \dots, (x^N, y^N).$$

We assume that we have a model of the form

$$y = f(x|\theta) = \sum_{n=1}^{N} \theta_f \phi_f(x^n), \tag{1.1}$$

where $\phi_f(x)$ is known as a "feature" which is some function of the input data x. Given the features (chosen manually) our optimization problem is given by

$$\min_{\theta} \sum_{n=1}^{N} (y^n - f(x^n|\theta))^2. \tag{1.2}$$

I would start by deriving the well-known normal equations, given by

$$\theta^* = [X^T X]^{-1} X^T Y,$$

where X is the *design matrix* given by

$$X = \begin{bmatrix} x_1^1 & x_2^1 & \dots & x_p^1 \\ x_1^2 & x_2^2 & \dots & x_p^2 \\ \vdots & \vdots & \vdots & \vdots \\ x_1^n & x_2^n & \dots & x_p^n \end{bmatrix},$$

and Y is our vector of observations (also called responses or labels)

$$Y = \begin{bmatrix} y^1 \\ y^2 \\ \vdots \\ y^n \end{bmatrix}.$$

This should be a warmup using their linear algebra and setting up a quadratic optimization problem that can be solved analytically.

Bring out that we need n observations $\geq p$ (number of parameters), and that the data has to be well behaved (so that the $[X^T X]$ matrix is invertible). It is easy to illustrate this with a simple example – fitting a line through two datapoints.

In the teaching notes below, we argue that that the optimization problem in (1.2) is actually a stochastic optimization problem. It looks deterministic here (and it is) because we are working with a sample of the random variables y. The sequence $(y^1, \ldots, y^N)$ is actually a sample of the random observations. We are going to see later that we can turn a lot of stochastic optimization problems into problems requiring deterministic methods by using samples, so this starter problem not only introduces an important application (machine learning), it is setting the stage for how we are going to solve a wide range of sequential problems that involve random information.

1.2 Nonlinear Models

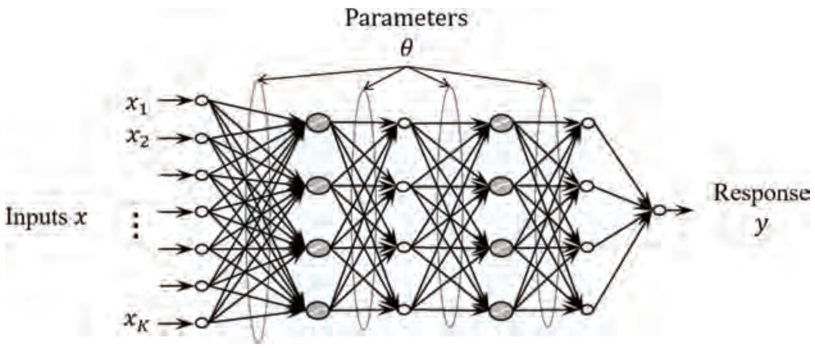
Next we transition from a linear model (where $f(x|\theta)$ is linear in the parameters) to a nonlinear model. One example is a logistic regression such as

$$f(x|\theta) = \frac{e^{\theta_0|\theta_1 x}}{1 + e^{\theta_0|\theta_1 x}},$$

or

$$f(x|\theta) = \begin{cases} -1 & x < \theta_1 \\ 0 & \theta_1 \leq x \leq \theta_2 \\ +1 & x > \theta_2 \end{cases}.$$

Or, our nonlinear model could be a neural network with millions (or billions) of parameters θ that looks like



We now have the optimization problem

$$\min_{\theta} g(\theta) = \sum_{n=1}^{N} (y^n - f(x^n|\theta))^2. \tag{1.3}$$

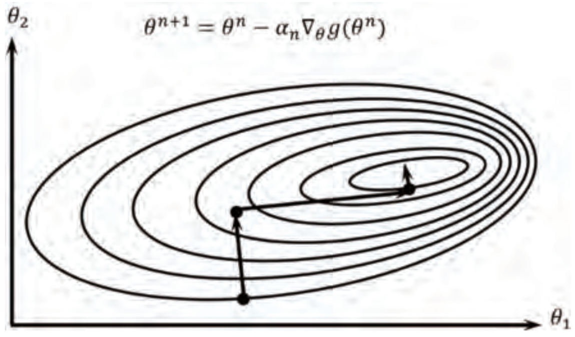
We can solve this using a gradient-based search algorithm that looks like:

$$\theta^{n+1} = \theta^n - \alpha_n \nabla_x g(\theta^n). \tag{1.4}$$

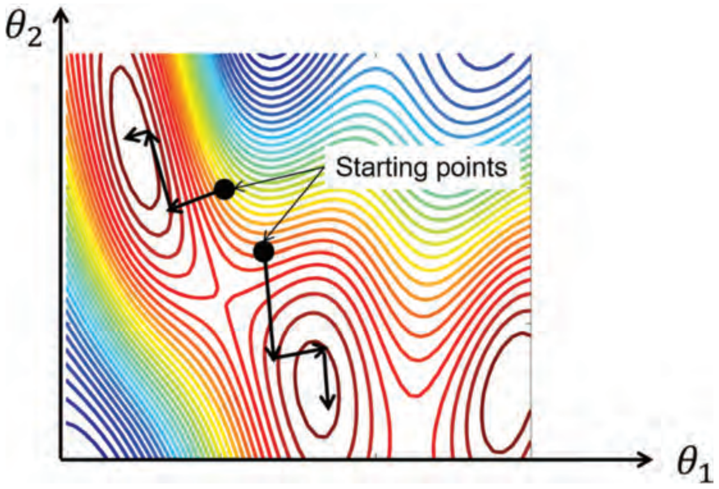
(Note that we use the negative gradient because we are minimizing). You need to talk the students through the process of finding the gradient (and possibly explaining what this is). One key step is finding the stepsize α_n which is done by solving the one-dimensional search problem:

$$\alpha_n = argmin_{\alpha>0} g(\theta^n - \alpha_n \nabla_x g(\theta^n)). \tag{1.5}$$

The figure below illustrates the search process.


θ_2
$\theta^{n+1} = \theta^n - \alpha_n \nabla_\theta g(\theta^n)$
θ_1

One issue that often arises is that the function $g(x|\theta)$ may have local minima (llustrated below), which means your gradient algorithm may produce different optimal depending on the starting point.



Also – I would note that while we need n (the number of observations) $\geq p$ (the dimensionality of θ) to use the normal equations for our linear model (in Topic 1), we no longer require this for the nonllinear model. We can apply the gradient algorithm for any value of n, even if it is smaller (and potentially much smaller) than p. Students need to understand that just because the algorithm returns an estimate of the best value of θ, that does not mean that it is guaranteed to be a good value that will work well on future datasets.

You can start by illustrating it using a linear model, and then show how to use it for a nonlinear model. For example, fit a logistic regression to predict demand as a function of price, or the probability of winning a bid for placing an ad on Google or Facebook. Then extend to a simple neural network. Be sure to highlight the presence of multiple optima and the need to use multiple starting points. Then, show that you get an answer even when $n < p$, an issue that becomes important with deep neural networks.

This will create a basis for later discussing a well-known issue with neural networks (all students will have heard of this as “AI”) which use models where $p \gg n$. Students will also learn that there is not a unique solution when we optimize parameters for nonlinear models. Finally, I would also make the point that while linear models require $n \geq p$, nonlinear models do not. We can fit a neural network with 100 million parameters with a single datapoint (!).

1.3 Neural Networks (Optional)

A special kind of nonlinear model is a neural network. Given the visibility of this technology, an instructor may wish to introduce students to the basic idea of a neural network, since the calculations are relatively simple. It is easy to show how to construct a basic neural network, and how to take a vector of inputs and translate it to an output (or set of outputs). Section 3.9.3 of RLSO describes how to compute the output of a neural network through a simple forward pass.

You can then jump to section 5.5 of RLSO if you wish to show how to compute the gradient of a neural network with respect to the weights on each link (these are the tunable parameters). Key to neural networks is that these derivatives are easy to compute. Given the interest in deep neural networks, you can show how these calculations can be done in parallel, which is why chips such as those by Nvidia (which are designed for massively parallel computation, originally for the graphics in video games) are so popular.

I would also point out that there are packages such as TensorFlow that do these calculations very effiiciently. However the calculations are performed, the core ideas are the same as what we illustrated using the logistic regression example in section 1.2. Remember – the idea in an introductory course is not to prepare students to actually do research on this topic; it is only to introduce them to important optimization problems and how we go about solving them. For example, the issue of multiple local optimal solutions that we introduced in our section on nonlinear models above is important, since it helps to understand that the "optimal" solution we obtain is not truly optimal, and that small changes in inputs might result in finding a different local solution.

Also – remember that while we require $n \geq p$ in a linear model, we have no such requirement for nonlinear models, including neural networks. We might have a neural network with 100 million parameters, but we will still get a number if we try to train it with just one data point. While this seems silly, it is quite common to fit neural networks where the number of observations (that is, the size of the training dataset) is quite a bit smaller than the number of parameters.

I don't personally encourage including this material, but it fits very nicely here, and might go a long way to attracting student interest.

1.4 Teaching Notes for Machine Learning

There are some key points that should be brought out in this topic:

- The linear model is a nice opportunity to remind students of some basic linear algebra when deriving the normal equations.
- Be sure to bring out the requirement that $[X^T X]$, where X is the "design matrix" of data, must be invertible, which requires that $n \geq p$. I suggest illustrating with the problem of fitting a line to a single data point, and then to two data points where x is the same for each one.
- The objective functions (equations (1.2) and (1.3)) look like deterministic optimization problems, but they are not. Fitting a function (linear or nonlinear) to data is actually a stochastic optimization problem which should be written

  $$\min_{\theta} \mathbb{E}_X \mathbb{E}_{Y|X} \left(Y - f(X|\theta)\right)^2. \tag{1.6}$$

  Here we view X, the explanatory variables (also known as independent variables or covariates), and the response $Y = f(X|\theta)$, as random variables. We assume we are given a sample of these variables

  $$(x^1, y^1),\ (x^2, y^2), \ldots, (x^N, y^N),$$

  which we call the training dataset. However, this is just a sample of the random variables X and Y, which turns our stochastic optimization problem (1.6) into a deterministic optimization problem (1.2) or (1.3). We are going to use this technique over and over again in this course to handle virtually any form of uncertainty. The only difference will be in future applications is that we may need a way of generating our own random sample.

Topic 2: Sequential Decision Problems I

The vast majority of all decision problems are sequential decision problems. Later, we are going to motivate harder decision problems that use linear, nonlinear and integer programming, but to start we are going to use simpler problems which are still important as well as challenging.

2.1 Introduction to Sequential Decision Problems

Readings: SDAM Chapter 1 (introduction to sequential decision problems).

A sequential decision problem is any problem that consists of the sequence:

decision, information, decision, information, . . .

where each decision receives a contribution or incurs a cost. Sequential decision problems cover an extremely broad class of optimization problems. Most important is that they cover problems that arise in almost any setting: business, health (all kinds), energy, economics, laboratory experiments, field experiments, . . . , the list is endless, which means it is possible to illustrate these problems with applications that are suitable to any class.

Decisions are made with methods that we call "policies." There are two broad strategies for designing policies, and each of these produces two classes, creating four classes of policies:

Strategy 1 – Policy search – This is where we identify a class of functions for making decisions, and then search for the best function that works well over time. The two classes of policies in this strategy are:

1. Policy function approximations (PFAs) – These are analytical functions that take what we know to determine what decision to make now. Examples are order-up-to policies for inventory, or buy low, sell high policies in finance.

2. Cost function approximations (CFAs) – These are simplified (usually deterministic) optimization models that have been parameterized to work well under uncertainty. We will see CFAs when

we introduce optimal learning (Topic 4). We will also see these when we introduce linear, integer and nonlinear programming in topics 7–11.

Strategy 2 – Lookahead approximations – We estimate the value of a decision by combining the immediate cost or reward of a decision plus some approximation of downstream costs and rewards from the initial decision. This strategy can be divided into two classes:

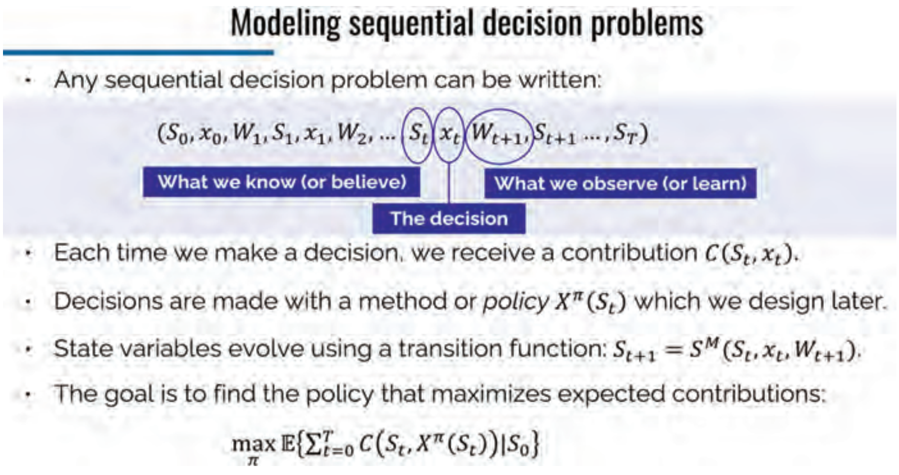
3. Value function approximations (VFAs) – Here we find the decision that optimize the immediate cost or reward plus an approximate value of the state that the decision takes us to. These are the only policies that use Bellman's equation (Hamilton-Jacobi if you are a controls person).

4. Direct lookahead approximations (DLAs) – Finally we optimize the immediate cost or reward plus an estimate of downstream costs or rewards computed by solving an approximate model of the future.

Note that sequential decision problems arise throughout human activities. PFAs are the simplest class and are the most widely used. Most important, these are parameterized functions, just like the parametric models in statistics that we saw in Topic 2.

Below is the slide I use to present the elements of a sequential decision problem. It illustrates the notation of states S_t (what we know at time t), decisions x_t (what decision we choose from a set of feasible decisions), and the exogenous information W_{t+1} that we learn only after we make the decision x_t. Decisions are made with a method (policy) that we designate as $X^{\pi}(S_t|\theta)$ that often depends on tunable parameters θ. Also shown is the contribution (if we are maximizing) or cost (if minimizing) $C(S_t, x_t)$ which may depend on information in the state S_t (such as dynamically changing prices or costs) in addition to the decision x_t. The transition function $S^M(S_t, x_t, W_{t+1})$ gives the updated state S_{t+1} given the information in S_t, the decision x_t, and the exogenous information W_{t+1}.

The next slide (below) is one I use to compare machine learning with sequential decisions. The difference between machine learning



Modeling sequential decision problems

- Any sequential decision problem can be written:

$$(S_0, x_0, W_1, S_1, x_1, W_2, \dots, S_t, x_t, W_{t+1}, S_{t+1}, \dots, S_T)$$

What we know (or believe)

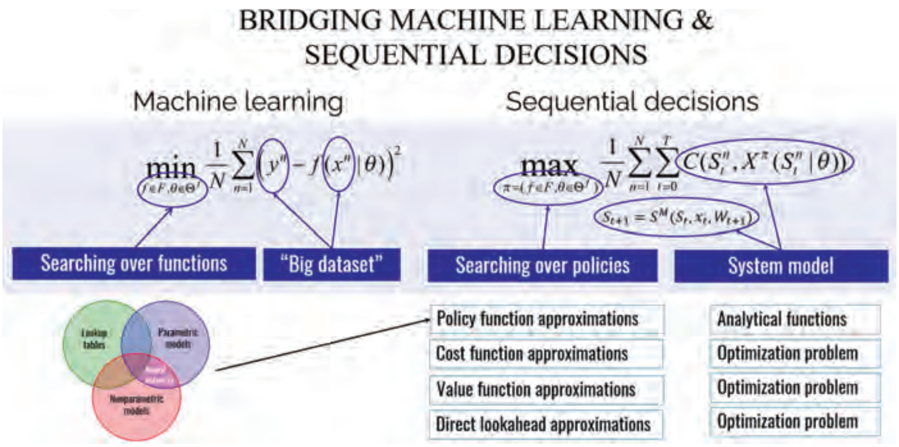
The decision

What we observe (or learn)

- Each time we make a decision, we receive a contribution $C(S_t, x_t)$.
- Decisions are made with a method or *policy* $X^\pi(S_t)$ which we design later.
- State variables evolve using a transition function: $S_{t+1} = S^M(S_t, x_t, W_{t+1})$.
- The goal is to find the policy that maximizes expected contributions:

$$\max_\pi \mathbb{E}\{\sum_{t=0}^T C(S_t, X^\pi(S_t))|S_0\}$$

and sequential decisions is that with machine learning, you need a training dataset to fit a general-purpose function, whereas sequential decisions (which do not need a training dataset) need a model of the underlying problem (such as the evolution of inventories). It is important to emphasize this difference, since it is often overlooked (especially in discussions of "AI").



BRIDGING MACHINE LEARNING &
SEQUENTIAL DECISIONS

Machine learning

$$\min_{f\in F,\theta\in\Theta^f} \frac{1}{N}\sum_{n=1}^{N}(y^n - f(x^n|\theta))^2$$

Searching over functions

"Big dataset"

Sequential decisions

$$\max_{\pi=(f\in F,\theta\in\Theta^f)} \frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{T} C(S_t^n, X^\pi(S_t^n|\theta))$$

$$S_{t+1} = S^M(S_t, x_t, W_{t+1})$$

Searching over policies

System model

Policy function approximations	Analytical functions
Cost function approximations	Optimization problem
Value function approximations	Optimization problem
Direct lookahead approximations	Optimization problem

There is a wide range of sequential decision problems that will be familiar to students (unlike linear programs). The challenge is that sequential decision problems involve the arrival of new information, which puts even simple decision problems into a class that has been treated under a wide variety of names, with the most common being "dynamic programs." This will produce gasps of "oh, that is *stochastic optimization...* you can't teach *stochastic optimization* in an introductory optimization course." For most faculty, "dynamic programming" means teaching Bellman's equation.

We are not going to go down that path. First, we observe that both of the machine learning problems in Topics 1 and 2 are forms of stochastic optimization problem. We avoid the complexities that would normally arise by tuning our statistical models used a training dataset that we assumed was given to us.

We are going to follow the same strategy for sequential decision problems, emphasizing three types of policies that are widely used, and which students will typically be familiar with. These are:

- PFAs – These are the simplest policies such as buy low, sell high in finance or order-up-to policies for inventory planning. Note that PFAs include every possible model that might be used in machine learning, which might be anything from a linear model to a deep neural network.

- CFAs – These will be simplified (always deterministic) optimization problems that might involve nothing more than a sort. For more complex problems, we will introduce linear programs, and then show how we can parameterize a linear program to make it work well over time. This powerful idea is widely used in practice but has been completely ignored in the academic literature.

- Deterministic DLAs – Google maps is a nice example of a deterministic DLA, since it plans a path to a destination (which requires looking into the future). For problems where we need to plan into the future, the most widely used strategy is to use a deterministic approximation. We *might* use tunable parameters to make these perform better over time, but not always.

Below we are going to illustrate some simple sequential decision problems using PFAs, which require tuning parameters. The approach exactly parallels what we did in Topic 1 for machine learning.

2.2 Asset Selling

Readings: SDAM Chapter 2.

We are going to start with a simple buy-low, sell-high policy in finance, which offers an important simplification to help us get started: the random information (the price at which we can sell our asset) is drawn from history, allowing us to treat it just as we did the observations y^n in our machine learning problems in Topic 1.

We are going to assume that we are given a historical sequence of prices

$$(p_{t-H}, p_{t-H+1}, \ldots, p_{t'}, \ldots, p_t)$$

where time t' is any point in time in the history where we might sell an asset (assume for simplicity that we can only sell at the end of each day). The price p_t would be the most recent price we have available. A nice feature of this problem is that we can assume that the prices are independent of any decisions that we make.

Next imagine that we are going to use the following policy that determines when to buy, sell or hold a single asset

$$X^{\pi_1}(S_t|\theta) = \begin{cases} +1 & \textit{Sell if } p_t > \overline{p}_{t-1} + \theta_1 \\ 0 & \textit{Hold if } \overline{p}_{t-1} - \theta_2 \leq p_t \leq \overline{p}_{t-1} + \theta_1 \\ -1 & \textit{Buy if } p_t < \overline{p}_{t-1} - \theta_2 \end{cases} \tag{2.1}$$

where

$$\overline{p}_t = .5p_t + .35p_{t-1} + 0.15p_{t-2} \tag{2.2}$$

is a smoothed estimated of prices. The variable S_t, which is called the "state variable" captures all the information we know at time t, that we need to make a decision (that is, compute our policy $X^{\pi_1}(S_t|\theta)$), as well as any other information we might need. For this problem, the state variable consists of

$$S_t = (p_t, p_{t-1}, p_{t-2}).$$

It is easy to see that our policy $X^{\pi_1}(S_t|\theta)$ is one of many possible strategies we could use. This policy has two tunable parameters. To tune the parameters, we need an objective function.

The objective function we would use to perform our tuning would look like

$$F(\theta) = \sum_{t'=t-H}^{t} X^{\pi}(S_{t'}|\theta)p_{t'}. \tag{2.3}$$

Note that our policy depends only on information that we would know at time t, even though we have an entire history of prices $(p_{t-H}, p_{t-H+1}, \ldots, p_{t'}, \ldots, p_t)$. This type of tuning is widely used in finance, and is known as "backtesting" since it requires using historical prices to evaluate the policy.

This objective closely parallels the one we used to fit nonlinear models in Topic 1. If we let $\theta_1 = \theta_2$, then we have a one-dimensional problem, which could be easily optimized in a spreadsheet (it is not much harder to do this over two dimensions).

This exercise accomplishes several educational objectives:

- It introduces the idea of a policy for making decisions.
- It uses historical data to represent a sample of outcomes, just as we created a training dataset for machine learning (see Topics 1 and 2).

A different way to perform the tuning is to create a mathematical model of prices, and use this to create a brand new set of prices for each iteration of the algorithm. We do not have to do this for this specific problem, although the tuning would be more robust if we did. However, creating mathematical models of prices is relatively difficult – we would need to capture both the distribution of prices (which is fairly easy) in addition to the correlations of prices over time (this is quite difficult). Working with historical prices avoids these complications.

2.3 Inventory Planning

Reading: Section 1.3 of SDAM

Inventory problems are one of the most popular applications of sequential decision problems. This will be a fairly minor variation of the asset selling problem, but it opens the door to an incredibly rich set of applications that arise in supply chain management.

For this topic, we will start with a vanilla inventory problem where we define:

$R_t =$ The amount of inventory at time t.

$x_t =$ The amount of new inventory we order, where we assume it arrives right away.

$\hat{D}_{t+1} =$ The demand for product that arises between t and t+1 (and after we make the decision x_t).

$p =$ The unit price at which we sell the product.

$c =$ The unit cost of ordering more product.

The basic equation for updating the inventory R_t is given by

$$R_{t+1} = \max\{0, R_t + x_t - \hat{D}_{t+1}\}. \tag{2.4}$$

Unlike our asset selling problem, we generally do not get to observe the actual demands $\hat{D}_t$, which means we cannot just use a set of observations from history. Instead, we are going to need to generate a set of observations of demands using a random number generator. We are going to take advantage of a standard function built into all computer languages to generate a random number between 0 and 1. For example, in Excel this function is called RAND(). In Python it is called Random.uniform(0,1). We are going to just let the function be represented by U().

If we want a random number R that is uniformly distributed between a and b, we use

$$R = a + (b - a)U(). \tag{2.5}$$

Now imagine that we know that on average the demand is μ, where our actual demand $\hat{D}_t$ is given by

$$\hat{D}_t = \mu + \varepsilon, \tag{2.6}$$

where ε is a random error term that is uniformly distributed between $-\mu$ and $+\mu$. We can generate random observations of ε using

$$\varepsilon = -\mu + 2\mu\, U(). \tag{2.7}$$

If we want 100 observations of demands, we need to generate 100 observations of ε.

Now that we have a set of random demands $\hat{D}_1, \hat{D}_2, \ldots \hat{D}_t, \ldots, \hat{D}_T$, we need a method for making decisions. We are going to use a policy known in the literature as an "(s, S)" policy where s and S are tunable parameters. We like to use θ for our tunable parameters, so we are going to replace s and S with θ^{min} and θ^{max}. Our policy is then given by

$$X^{\pi}(S_t|\theta) = \begin{cases} \theta^{max} - R_t & if\ R_t < \theta^{min} \\ 0 & otherwise. \end{cases} \tag{2.8}$$

Our challenge now is to find the best value of $\theta = (\theta^{min}, \theta^{max})$, which we do by optimizing the objective function

$$\max_{\theta} \sum_{t=0}^{T} p\ max\{R_t + X^{\pi}(S_t|\theta), \hat{D}_{t+1}\} - cX^{\pi}(S_t|\theta), \tag{2.9}$$

where the inventory R_t evolves according to equation (2.4).

The optimization problem in (2.9) can be approached just as we did to optimize our nonlinear model in Topic 1 (equation (1.3)) or the asset selling problem in section 2.2.

2.4 Teaching Notes

- We have now seen that optimizing a parameterized policy closely parallels optimizing the parameters of a nonlinear function for machine learning. One difference is that we can generally compute the gradient of the nonlinear function in machine learning, whereas this is typically not true when we are simulating a policy.

- The simplest approach to use right now is to introduce the idea of numerical derivatives.

- There is also a wide range of derivative-free methods, but these should be introduced slowly over the course, depending on the interests of the professor and the backgrounds of the students.

Topic 3: Adaptive Optimization – The Newsvendor Problem

Readings: Chapter 3 in SDAM

Arguably one of the most widely encountered problems when managing resources is the newsvendor problem, which is a nice illustration of a decision problem involving uncertainty. We will show how to solve this using a simple stochastic gradient algorithm that can be implemented in an online (learn as you go) setting. This is a natural extension of the gradient-based method we used in section 1.2 for optimizing the parameters of a nonlinear statistical model.

In Topic 1, we solved the following machine learning problem

$$\min_\theta \overline{F}(x|\theta) = \frac{1}{N}\sum_{n=1}^{N}(y^n - f(x^n|\theta))^2. \tag{3.1}$$

This is a sampled estimate of the function

$$\min_\theta F(x|\theta) = \mathbb{E}(Y - f(X|\theta)), \tag{3.2}$$

where "Y" is a random variable representing the response given the random input "X", and where

$$(x^1, y^1),\ (x^2, y^2), \ldots (x^n, y^n), \ldots (x^N, y^N),$$

is a sample of N observations of the variables (X, Y).

The idea of using a sampled estimate to transform a stochastic optimization problem (3.2) into a deterministic optimization problem (3.1) is a powerful and widely used strategy when optimizing functions of random variables.

One of the most popular stochastic optimization problems is the *newsvendor problem*, where we need to choose a quantity x to meet an unknown demand D. The challenge is that we have to purchase x units of a product at a unit cost c, to meet the demand D receiving a revenue p for each unit sold. The problem is that we cannot sell more than the demand, giving us the objective function

$$\max_x F(x|D) = p\max\{x, D\} - cx. \tag{3.3}$$

The function $F(x|D)$ assumes we know the demand D, but we do not. What we have to do is to find the quantity x that maximizes the expected value of $F(x|D)$ over the random quantity D. This would be written

$$\max_x g(x) = \mathbb{E}_D F(x|D) = \mathbb{E}_D\{p \max\{x, D\} - cx\}. \tag{3.4}$$

Imagine that we have a historical dataset of order quantities and demands which we can write out as

$$(x^1, D^1),\ (x^2, D^2), \ldots (x^n, D^n), \ldots (x^N, D^N).$$

If we have this data, we could solve our problem just as we did our machine learning problem in (3.1):

$$\max_x \overline{F}(x) = \frac{1}{N}\sum_{n=1}^{N} (p \max\{x, D^n\} - cx). \tag{3.5}$$

The problem with this approach is that we never have a set of observations of demands $D^1, D^2, \ldots, D^N$ because we do not observe demands – we observe what we sell which is the smaller of what we ordered x^n and the true demand D^n. If we order $x^n = 6$ and observe sales of 6, it might be that $D^n = x^n$, but more often it means that $D^n > x^n$.

There is a simple way to get around this problem. We are going to use a basic gradient search algorithm, but we cannot take the derivative of the function $g(x)$. What we are going to do seems magical (that is, it seems as if we should not be able to do it). We are going to assume that we choose a quantity $x = x^n$, and then observe a demand D^{n+1}. Note that we have introduced a subtle shift in how we are indexing x^n and the demand D^{n+1}; this way of indexing means that x^n depends on $D^1, \ldots, D^n$ but does not depend on D^{n+1}.

After we observe the demand, we now have the deterministic function

$$g(x|D^{n+1}) = p \min\{x, D^{n+1}\} - cx. \tag{3.6}$$

Next we are just going to take the derivative of $g(x|D^{n+1})$ with respect to x:

$$\nabla g(x|D^{n+1}) = \frac{dg(x|D^{n+1})}{dx} = \begin{cases} p - c & if\ x \le D^{n+1} \\ -c & if\ x > D^{n+1} \end{cases}. \tag{3.7}$$

We now use the same gradient-based search algorithm we first introduced in Topic 1 for nonlinear models:

$$x^{n+1} = x^n + \alpha_n \nabla g(x|D^{n+1}). \tag{3.8}$$

Unlike our first use of gradient based methods in Topic 1, we can no longer find the stepsize α_n by solving a one-dimensional search problem as we did in equation (1.5). Instead, we are going to use something that is much simpler:

$$\alpha_n = \frac{1}{n}. \tag{3.9}$$

Incredibly, we can show that this stepsize rule will produce a sequence of decisions $x^1, x^2, \ldots, x^n$, where

$$\lim_{n\to\infty} x^n \to x^*. \tag{3.10}$$

This means that if we run this algorithm an infinite number of times, it will find the optimal solution! The bad news is that it is possible that the algorithm will be quite slow. A lot of research has gone into finding better stepsize formulas. One way speed up the algorithm is to insert a tunable parameter giving us

$$\alpha_n(\theta^{step}) = \frac{\theta^{step}}{\theta^{step} + n - 1}. \tag{3.11}$$

where θ^{step} is a parameter that has to be tuned. OK, so we have another problem that involves tuning a parameter, but the core idea here is quite simple!

Stochastic gradient algorithms tend to be taught in advanced stochastic optimization classes. However, they are perfectly appropriate for an introductory optimization course, especially for the context of unconstrained problems. Nonnegativity constraints and upper bounds are easy to handle.

Topic 4: Optimal Learning – Finding the Best Treatment

Readings: Chapter 4 in SDAM

An important class of optimization problems falls under the umbrella of "optimal learning" where the decision involves what to observe, or what experiment to run. From the information gained from the observation (or experiment). We then use our beliefs to make a choice about a choice of design, or product, or price.

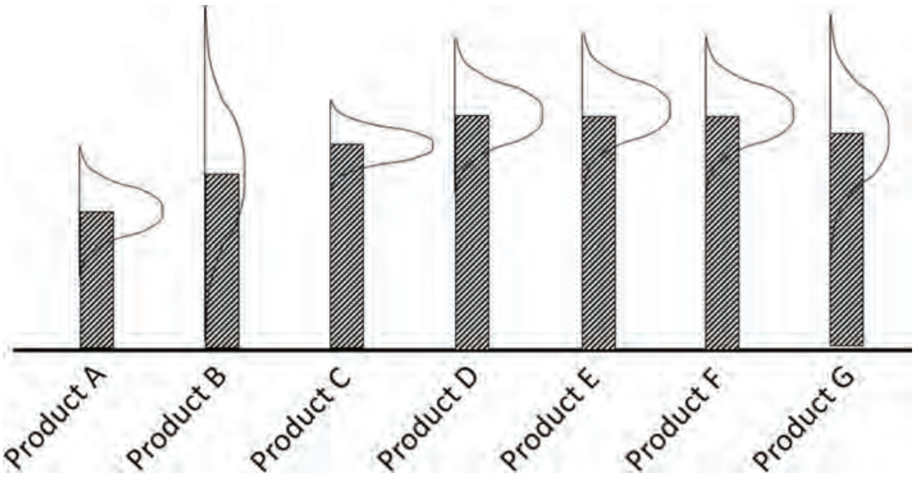
There is an endless array of optimal learning problems – a sample might include:

- Which medical treatment to use (choice of drug, dosage)
- Which product to advertise on a website
- Which supplier to use to supply materials or components
- Which schools to visit to interview for employees
- What price to charge for a product (from a set of prices)
- Who should be starters on a basketball team
- Which path to take through a congested network to get to work
- Which product to recommend on a webpage to attract the most clicks
- Which of several webpage designs to use to maximize traffic
- ... (This is a very long list)

In this section we will use the context of finding the best medical treatment for a patient.

Learning the best of a set of "treatments" is widely known as a "multiarmed bandit problem" which is a mathematically rich (and computationally complex) problem. However, the approach we are going to use here is quite simple and very popular at companies like Google and Facebook for optimizing ads (students should resonate with this). In the process, we are also going to introduce a new class of policy that is going to open the door to a much more complex class of problems. These policies are known as "cost function approximations" (or CFAs).

We are going to start with a policy called "interval estimation" that helps us choose the best treatment $x \in X = \{x_1, \ldots, x_K\}$. Let $\overline{\mu}_x^n$ be our estimate of the effectiveness of treatment x after we have performed n experiments using any of the choices. Then let $\overline{\sigma}_x^n$ be the standard deviation of $\overline{\mu}_x^n$ (remind students how to do the standard deviation of a mean, and how it goes to zero as $n \to \infty$).



After n experiments, our beliefs might look like those shown in the figure above – this might represent the potential sales of different products, but there are *many* settings where this applies. We represent our beliefs using the belief state variable $B^n = (\overline{\mu}_x^n, \overline{\sigma}_x^n)_{x \in X}$ which capture our beliefs about the performance of each choice x. For this problem, the state variable $S^n = B^n$ equals the belief state, but there are problems where we may have information other than the belief state (such as the budget remaining to run experiments). For now, we are just going to limit the state variable to the belief state.

The interval estimation policy is given by:

$$X^{IE}(S^n|\theta^{IE}) = argmax_{x \in X}(\overline{\mu}_x^n + \theta^{IE}\overline{\sigma}_x^n) \qquad (4.1)$$

where $x^n = X^{IE}(S^n|\theta^{IE})$ is the design we are going to choose for the $n + 1^{st}$ experiment, which produces an observation W^{n+1}.

With our inventory problem, we updated our state variable (the inventory) using the inventory equation (2.4). With a learning problem, we have to update our beliefs, which we are going to do using some simple recursions. Assume that we observe performance W_x^{n+1} when we test choice $x = x^n$. First, we are going to replace the variances $(\overline{\sigma}_x^n)^2$ with their inverses which we call the *precision* given by

$$\beta_x^n = \frac{1}{(\overline{\sigma}_x^n)^2}. \tag{4.2}$$

We are also going to assume that when we observe the results of an experiment which we represent by W_x^{n+1} that this experiment is random with a known variance σ_W^2 and precision

$$\beta^W = \frac{1}{\sigma_W^2}. \tag{4.3}$$

We can use the precision to write the updating equations for the means and precisions using

$$\overline{\mu}_x^{n+1} = \frac{\beta_x^n \overline{\mu}_x^{n+1} + \beta^W W_x^{n+1}}{\beta_x^n + \beta^W}, \tag{4.4}$$

$$\beta_x^{n+1} = \beta_x^n + \beta^W \ . \tag{4.5}$$

Equations (4.4) and (4.5) represent the transition equations $S^{n+1} = S^M(S^n, x^n, W^{n+1})$ for this problem.

An important feature of our interval estimation policy (4.1) is that imbedded in the policy is an optimization problem. Here the optimization (given by the "argmax") requires nothing more than a simple sort over the alternatives to find one with the best value of $\overline{\mu}_x^n + \theta^{IE}\overline{\sigma}_x^n$. Later, we are going to replace this with more sophisticated optimization problems. For example, in Topic 5, our imbedded optimization problem will be a shortest path problem. In Topic 7 the imbedded optimization problem will be a linear program, which we see again in Topic 8 when we are dynamically planning energy storage. In Topic 10 the imbedded optimization problem will be an integer program, and in Topic 11 (section 11.2) the imbedded optimization problem will be a nonlinear programming problem.

The interval estimation policy is very easy to implement, but it is important not to overlook the need to tune the parameter θ^{IE}. One way to do this is to run a simulation where we assume that we know the true performance of each alternative x which we denote by μ_x. If we choose to test alternative x, we cannot observe μ_x perfectly – instead, we can only perform a noisy observation where we add a noise term ε. This means that the observed performance of $x = x^n$ would be given by

$$W_x^{n+1} = \mu_{x^n} + \varepsilon^{n+1} \tag{4.6}$$

We can use the methods we presented for the inventory planning problem (see equations (2.5)–(2.7)) to generate random observations of ε. Next, create N (say, $N = 100$) observations of W_x^n for each alternative x and store these. Then, simulate our interval estimation policy $X^{IE}(S^n|\theta^{IE})$ which we evaluate using

$$F(\theta) = \sum_{n=0}^{N-1} W_{x^n}^{n+1} \tag{4.7}$$

where $x^n = X^{IE}(S^n|\theta^{IE})$ and where the state variable S^n is updated using equations (4.4) and (4.5). Note that we have written (4.7) as if we are running a single simulation. We can do this, but it will be noisy. Instead of pre-generating the outcomes of W_x^n and using these in the simulation of the policy, it makes more sense to generate them on the fly (this is very fast using any programming language such as Python). Now compute the sum in (4.7), say, 1000 times and take an average. Finally, repeat this for a discrete set of values of θ such as 0, 0.1, 0.2,. . . , 4.0 and choose the value of θ^{IE} that works the best.

Topic 5: Shortest Path Problems

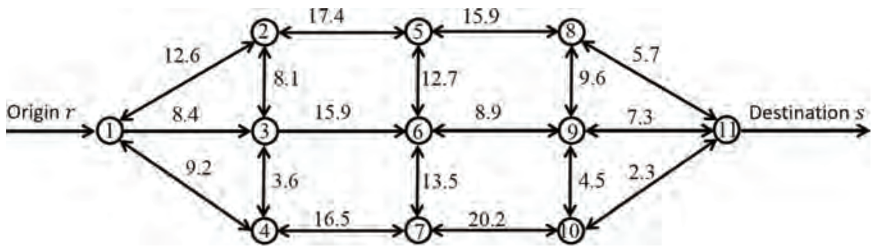
We are going to use shortest path problems as our first small step into a nontrivial decision problem. While this is technically a linear program, we will be able to demonstrate a simple algorithm based on Bellman's equation that everyone can understand. Shortest path problems also provide a nice visual setting to later illustrate the idea of a basis and pivoting (but this is for later).

We are going to first present the static shortest path problem, and then transition to using this in a dynamic setting as would occur when you are driving from one location to the next while your navigation system is responding to new information. It is in the dynamic setting that we will see that a shortest path problem is a form of policy (specifically a direct lookahead policy, or DLA) that can, if we like, be parameterized to help deal with uncertainty.

5.1 Static Shortest Paths

This lecture is based on chapter 5 of SDAM.

This will be our first peek at a specialized linear program that we will solve using a classical Bellman iteration. This is very simple in the context of a network problem since the "state" variable is just the node where the traveler is located.



Assume we are trying to find the shortest path from origin $s = 1$ to destination $r = 11$. Let

v_i = the minimum cost to get from node i to the destination node $r = 11$. We are going to initialize $v_{11} = 0$, and set v_i equal to some large number for all other nodes $i \neq 11$.

Let

N_i^+ = the set of nodes we can reach from node i.

A simple (but not very efficient) algorithm for finding the shortest path from each node to node 11 would be to repeatedly compute for every node i

$$v_i = \min_{j \in N_i^+} (c_{ij} + v_j). \tag{5.1}$$

The idea is to repeatedly loop over all nodes i and compute (5.1) until none of the node values change. We can store the optimal solution by letting

$$x_{ij}^* = \begin{cases} 1 & if\ j = \operatorname{argmin}_{k \in N_i^+}(c_{ik} + v_k) \\ 0 & Otherwise \end{cases} \tag{5.2}$$

Equation (5.1) is known as *Bellman's equation* and is very popular in the academic literature for solving a wide range of sequential decision problems. In practice, it only works for a very small subset of problems, but this happens to be one where it works very well (although commercial algorithms use a lot of shortcuts).

We can trace the shortest path by starting at node r and then traversing from any node i to the node j where $x_{ij}^* = 1$.

5.2 Dynamic Shortest Paths

Readings: SDAM Chapter 6

Here we introduce the problem faced by Google maps. We have to route a traveler through a network where new information is arriving over time. Imagine that time steps forward one increment each time we traverse a link. When the traveler arrives at node i at time t, Google receives updated estimates of travel times and recomputes the shortest path. This is a dynamic system where the "state" now includes two pieces of information: the node where the traveler is located (node i), and the updated estimates of the travel times over the *entire* network.

To keep the notation simple, assume that we get updates of estimated travel times at each time period, although in reality we only need to update the travel times each time a traveler arrives at a node and has to make a decision. Now, instead of a fixed cost c_{ij}, we have costs that are updated and depend on time t, so we define

c_{tij} = The estimated travel time on each link (i,j) given the information we have at time t.

We then solve the same static shortest path problem we did above, but instead we are using the updated costs c_{tij}, and obtain the updated path x^*_{tij}. We are not going to implement the entire shortest path – instead, if we are at some node i at time t, we are going to choose to go to node j if $x^*_{tij} = 1$.

We can write our shortest path problem at time t using our vocabulary of policies. At time t, our state variable S_t captures what we know, which includes the node i_t where we are located, and the current estimates of all the link costs which we can write as

$$c_t = (c_{tij}) \text{ for all links } (i,j) \text{ in the network.}$$

So we would write our state variable as

$$S_t = (i_t, c_t).$$

Our policy is to solve the static shortest path problem using our updated vector of costs c_t, but the policy only returns what the traveler should do at time t, which we can write as

$$X^\pi_t(S_t) = x^*_{i_t,j}.$$

This means that $X^\pi_t(S_t)$ is a vector of 0's with a 1 in the entry corresponding to $x^*_{i_t,j_t} = 1$. We will let

$$X^\pi_{ti_t,j}(S_t) = \begin{cases} 1 & if\ j = j_t \\ 0 & Otherwise \end{cases}$$

Let's introduce a twist. Imagine that we have a goal of reaching our destination by a particular time, and while the shortest path suggests that we will arrive in time, we recognize that there is uncertainty in the travel times. The costs (times) c_{tij} are just the estimated means from the sampled observations we have from watching individual travelers. Instead of using an average, what if we use the 80th percentile, or the 90th percent, or the 50th? These are easy to compute from the raw data.

Let

θ^{pctile} = the percentile of the travel time for a link.

$c_{tij}(\theta^{pctile})$ = the travel time corresponding to the θ^{pctile} of the travel times.

Now we use $c_{tij}(\theta^{pctile})$ for the travel times (instead of the means c_{tij}). We would then write our policy as $X^{\pi}_t(S_t|\theta^{pctile})$ to express the dependence on θ^{pctile}. The performance of the policy $X^{\pi}_t(S_t|\theta^{pctile})$ depends on both the actual travel time, but also how late the traveler is for their appointment. Typically we would add a penalty θ^{late} times how late the traveler.

Now we have another tuning problem just like we saw with PFAs (Topic 2), and solved using the same methods we saw in Topic 1. To evaluate our policy, let

$\hat{c}^n_{tij}$ = the sample realization of the actual time to traverse link (ij) that we reach at time t. These samples are not used to plan a path – they are only used to evaluate the policy for making decisions. We can generate $\hat{c}^n_{tij}$ using the Monte Carlo simulation methods we introduced in section 2.3.

Next let

$\hat{F}^n(\theta)$ = the actual travel time over the entire path for the n^{th} trial, using costs $\hat{c}^n_{tij}$,

$= \sum_{t=1}^T \sum_{ij} X^{\pi}_{tij}(S_t|\theta)\hat{c}^n_{tij}$.

$\hat{F}^n(\theta)$ is the actual travel time we experience in our n^{th} trip following policy $X^{\pi}_t(S_t|\theta)$ while experiencing link costs $\hat{c}^n_{tij}$. Let's say that we have to finish the trip in time τ to arrive in time for our appointment. Let

η = penalty per unit time for being late.

The total cost (time plus late penalty) for the n^{th} trip is then

$$\hat{C}^n(\theta) = \hat{F}^n(\theta) + \eta \max\{0, \hat{F}^n(\theta) - \tau\}. \tag{5.3}$$

We can then write the performance of our policy by averaging over N as

$$\overline{C}^{\pi}(\theta) = \frac{1}{N}\sum_{n=1}^{N} \hat{C}^n(\theta).$$

We now have another instance of needing to tune the parameter of a policy.

This lecture is setting the stage for parameterizing linear programs. When we present linear programs in Topic 7, we are going to start by presenting a basic static, deterministic linear program (just as we did with our initial shortest path problem), and then transition to recognizing that the linear program we chose is typically solved repeatedly over time, just as we have done above with our shortest path problem.

Topic 6: General Concepts

Up to now we have been approaching problems in what might appear an ad hoc manner. Actually our problems have been carefully chosen to illustrate some important dimensions of modeling and solving different types of decision problems.

We are going to step back and talk more generally about modeling, designing policies, and evaluating policies.

So far we have seen two types of optimization problems:

- Static problems (the machine learning problems in Topic 1, and the shortest path problem in Topic 5, section 5.1).
- Sequential decision problems (Topics 2, 3 and 4, and the dynamic shortest path problem in Topic 5, section 5.2).

Below we are going to provide modeling frameworks for each of these problems.

6.1 Modeling Static Optimization Problems

For our machine learning problems, we faced the problem of minimizing a nonlinear function (the sum of squares of errors) of a set of tunable coefficients θ. If we let

$$F(\theta) = \sum_{n=1}^{N} (y^n - f(x^n|\theta))^2, \tag{6.1}$$

then we can write our optimization problem as

$$\min_{\theta} F(\theta). \tag{6.2}$$

We refer to $F(\theta)$ as our objective function. The standard form that we might use would be to replace θ with x which is our standard notation for a decision variable, and let $C(x)$ be a generic "cost" function (assuming we are minimizing, which is what is typical in deterministic optimization). Our problem would then be written

$$\min_{x} C(x). \tag{6.3}$$

We can use this style for our shortest path problem, where $x = (x_{ij})$ is the vector of flows over the network. We would let

$$x_{ij} = \begin{cases} 1 & \textit{if link } (i,j) \textit{ is in the shortest path tree to the} \\ & \textit{destination.} \\ 0 & \textit{Otherwise} \end{cases}$$

We would then write our objective function as

$$C(x) = \sum_{i,j} c_{ij} x_{ij}, \tag{6.4}$$

where c_{ij} is the cost of traversing link (i, j). The problem with this formulation is that the optimal solution is $C(x) = 0$ since we would just set $x_{ij} = 0$. We need to introduce *constraints* so that we guarantee that our optimal solution returns a shortest path from the origin r to the destination s. We do this by introducing constraints which would be written

$$\sum_j x_{ij} - \sum_i x_{ij} = 0 \qquad \textit{for } i,\ j \neq r \textit{ or } s, \tag{6.5}$$

$$\sum_j x_{ij} - \sum_i x_{ij} = 1 \qquad \textit{for } i = r, \tag{6.6}$$

$$\sum_j x_{ij} - \sum_i x_{ij} = -1 \qquad \textit{for } i = s. \tag{6.7}$$

We then typically impose the requirement that x_{ij} cannot be negative by writing

$$x_{ij} \geq 0. \tag{6.8}$$

We often want to include upper bounds u_{ij}. In our shortest path problem, equations (6.5)–(6.7) would allow a solution where $x_{ij} > 1$ implying a path that is running in circles (which is clearly not a good idea). If we want upper bounds, we would then write

$$x_{ij} \leq u_{ij}. \tag{6.9}$$

Equations (6.5)–(6.7) can be written in matrix form

$$Ax = b \tag{6.10}$$

by suitably constructing the matrix A and the vector b. By doing this, we can now write our optimization problem in the form

$$\min_x C(x) = \sum_{ij} c_{ij} x_{ij} = c^T x, \tag{6.11}$$

subject to the constraints

$$Ax = b, \tag{6.12}$$

$$x \leq u, \tag{6.13}$$

$$x \geq 0. \tag{6.14}$$

Equations (6.11)–(6.14) is a fairly standard way of writing a static optimization problem. What is most important about this framework is that it is a *language* for *thinking* about a very large class of decision problems.

An important dimension of modeling optimization problems is the use of constraints such as (6.12)–(6.14). We first saw constraints in our shortest path problem, but these were enforced by the concept of finding a path. The optimization problems in Topics 1 and 2 were unconstrained. Linear programs (which we introduce in Topic 7) are meaningless without constraints. In fact, we are going to see that expressing problem characteristics through constraints is an art form that is a necessary skill for people who want to use linear programs.

Constraints represent a major feature of static optimization problems. They come in a number of flavors:

- Unconstrained problems, which we saw in Topic 1 for machine learning.

- Upper and lower bounds (also known as "box" constraints), such as (6.13) and (6.14).

- General linear constraints such as (6.12). With some creativity, we can handle a wide range of constrained problems using linear constraints. For our network problem, linear constraints are natural, but later (in Topic 10, section 10.3) we are going to need some creativity to express constraints using linear equations.

- Nonlinear constraints – An example might be $x_i(1 - x_j) = 0$. We do not deal with nonlinear constraints in this course.

Easily the most widely used algorithmic search strategy depends on gradients. The simplest search algorithms are gradient-based, as we saw in section 1.2 for optimizing the parameters of a nonlinear machine learning model, and again in topic 3 for our newsvendor problem. In Topic 7, we are going to see how to use a gradient-based algorithm in the presence of linear constraints using a method that is widely known as the simplex algorithm.

The literature on optimization problems is incredibly rich, but often ignores that an optimization problem is just solving a decision problem at one point in time, whereas the real application involves decisions that are made sequentially over time.

6.2 Modeling Sequential Decision Problems

Modeling any sequential decision problem starts by answering three questions:

1. What are the performance metrics?
2. What types of decisions are being made (and for larger problems, who makes each type of decision)?
3. What are the sources of uncertainty?

The answers to these three questions lay the foundation for building our model.

Any model of a sequential decision problem can be broken down into five components:

1. State variables S_t – A "state variable" S_t is all the information we need to model the system from time t onward. In other words, the state variable can be viewed as the "state of information" or, more precisely, the "state of knowledge," capturing everything we know or believe that we need to (a) compute the objective function (e.g. changing prices and costs), (b) make a decision, which means the

information needed to represent the available set of actions (e.g. what product to choose among those that are available) or the constraints that determine the feasibility of x, and (c) any other information needed to model the evolution of information in (a) or (b) (see the "transition function" below).

It is useful to distinguish between the initial state S_0 that might include static information that never changes (such as the maximum speed of a truck), from the dynamic state variables $S_t, t > 0$ which includes information that is changing over time such as the location of a vehicle moving over a network, the amount of inventory being held, or the evolving prices of an asset.

State variables come in three flavors:

- R_t = Vector of physical and financial aresources (people, product, equipment, facilities). This can be inventories, or the location of a person or piece of equipment that is moving around. It can also be the amount of cash on hand, investments of different kinds, loans, ...
- I_t = Other information about the system not included in R_t such as prices, weather, market conditions.
- B_t = Beliefs about quantities and parameters that are not known perfectly. This could be the mean and variance of a normal distribution, a vector of probabilities of discrete values of parameters such as a cost or constraint (we first saw belief state variables in Topic 4).

There is tremendous (and surprising) confusion about state variables in the academic literature. There are entire fields (Markov decision processes, reinforcement learning) that never even define a state variable. For a more in-depth discussion see the webpage

https://tinyurl.com/onstatevariables/

2. Decision variables x_t – These are the variables that we are controlling. We distinguish between initial decisions x_0 which are made

once (called design decisions), versus $x_t, t > 0$ which are made over time (called control decisions). We represent the feasible decisions by creating a set X_t that can be a feasible set of actions (e.g. what link to traverse, which person to hire) or the feasible region defined by constraints such as (6.12)–(6.14). Finally, we introduce a function $X^\pi(S_t|\theta)$, which we call a *policy*, which determines how we make a decision from the information available in the state variable S_t. Our policies usually depend on tunable parameters θ (but not always). Most important: *We will determine the policy later!*

3. Exogenous information W_{t+1} – This is information that we did not know at time t, but which became available by time *t+1* when we have to determine the decision x_{t+1}. In our examples above, the information $W_1, \ldots, W_t, \ldots, W_T$ is generated in advance, either from history (as we did with the asset selling example) or from Monte Carlo simulation, as we did in the inventory planning example. Although we created this information in advance, we never made a decision x_t using the information in W_{t+1} or later.

   Up to now, we have generated a single random sequence of the sequence of observations $W_1, \ldots, W_t, \ldots, W_T$, which we then used just as we used the training data in our machine learning problems in Topic 1. However, there are settings where W_{t+1} depends on the state variable S_t, or the decision x_t, or both. We can still use a sample of the sequence $W_1, \ldots, W_t, \ldots, W_T$, but we cannot generate it in advance. Instead, we have to generate it as the system evolves.

   Sometimes (in fact, frequently) we want to explicitly capture that there may be more than one sequence of exogenous information. Imagine that we are generating the demands for our inventory problem. Instead of generating one sample, we generate 10 as shown below. Following standard practice from the modeling literature, we let the Greek letter ω (“omega”) index the sample paths. So, if we generate 10 samples of the demands, ω would range from 1 to 10.

ω	D$_1$	D$_2$	D$_3$	D$_4$	D$_5$	D$_6$	D$_7$	D$_8$	D$_9$
1	18	16	13	10	17	6	4	15	16
2	12	7	17	15	5	3	4	14	8
3	6	18	7	9	1	13	4	4	7
4	2	11	16	16	1	2	13	0	13
5	18	5	0	6	10	17	8	3	2
6	3	18	5	20	13	16	18	11	10
7	12	14	4	11	19	3	20	19	18
8	6	15	15	14	2	7	14	1	11
9	19	10	5	19	13	14	16	11	17
10	18	15	14	4	6	17	16	10	9

To indicate a particular realization of W_t, we would write $W_t(\omega)$ (if we are using demands D_t, we would write $D_t(\omega)$). We can use this notation to compute, say, the average demand at time t using

$$\overline{D}_t = \frac{1}{10}\sum_{\omega=1}^{10} D_t(\omega). \tag{6.15}$$

4. Transition function – This is the function that describes how the information in the state variable evolves over time. We write this function using

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}). \tag{6.16}$$

In other words, given what we know (or believe) which is captured in S_t, the decision we made x_t, and the new information that arrived from outside the system (which is not known at time t), given by W_{t+1}, the transition function returns the updated state variable S_{t+1}. The notation $S^M(\cdot)$ stands for "state transition model" (or if you like, "system model").

A transition function can be a single equation such as the inventory equation (2.4) for our inventory planning problem. However, for complex systems (supply chains, trucking companies, energy systems, health systems) the transition function may require many thousands of lines of code.

5. Objective function – With deterministic optimization, objective functions are very straightforward, typically summing costs that are minimized, or it might be some performance metric to be maximized.

   With sequential decision problems, objective functions can come in different styles. We start by assuming that we are maximizing the cumulative contribution (or reward) over time, as we did in the problems in Topic 2. Let's start by writing the contribution in each time period as

$C(S_t, x_t) =$	the contribution from decision x_t made at time t, using the information in S_t such as a dynamically varying cost or price, such as the price p_t in our asset selling problem.

   We may write the contribution using $C(S_t, X^\pi(S_t|\theta))$ to reflect the dependence on the policy, since $x_t = X^\pi(S_t|\theta)$.

   Now imagine that we have created in advance a random sample of the information sequence $W_1, \ldots, W_T$ as we did in our asset selling example or the inventory planning example. Our objective function would then be written

   $$F(\theta) \approx \sum_{t=0}^{T} C(S_t, X^\pi(S_t|\theta)) \tag{6.17}$$

   where the transition function $S_{t+1} = S^M(S_t,\ x_t = X^\pi(S_t|\theta), W_{t+1})$ is computed with our sampled set of observations $W_1, \ldots, W_T$.

   If we wish to use more than one sample of $W_1, \ldots, W_T$, we can assume we have a set $W_t(\omega)$ for $\omega = \{\omega^1, \ldots, \omega^n, \ldots, \omega^N\}$. Now we would model our transition function by reflecting the dependence on the sample ω which we can write using

   $$S_{t+1}(\omega) = S^M(S_t(\omega), x_t(\omega) = X^\pi(S_t(\omega)|\theta), W_{t+1}(\omega)). \tag{6.18}$$

   We then calculate an estimate of our objective function using

   $$F(\theta) \approx \frac{1}{N} \sum_{n=1}^{N} \sum_{t=0}^{T} C(S_t(\omega^n), X^\pi(S_t(\omega^n)|\theta))] \tag{6.19}$$

Here we are just running N simulations and taking an average. However we compute the objective function, our optimization problem is then written

$$\min_{\theta} F(\theta) \tag{6.20}$$

Instead of listing constraints as we did in our static objective function (these are built into the design of our policy $X^{\pi}(S_t|\theta)$), we would follow the statement of the objective in (6.20) with the transition function (6.16) (or (6.20)) and the information given to the model in the form of the initial state S_0 and the information $W_1, \ldots, W_T$.

We see that modeling a sequential decision problem is much richer than a static optimization model, but it is roughly a mathematical statement of the simulations that we have already illustrated in Topic 2.

Our modeling framework can be used to model *any* sequential decision problem, although there are other choices for the objective function. Next we are going to turn to the issue of designing policies, which is a little richer than we have indicated above.

6.3 Designing Policies

Above, we wrote our policy as $X^{\pi}(S_t|\theta)$ which seems to imply that we already have some functional form for the policy, and then have to tune the parameters θ. However, just as we have to choose between different models in machine learning (such as the linear and nonlinear models we saw in Topic 1), we also have to choose among different functional forms for policies. However, the set of choices becomes much broader.

We can divide all the different types of policies into four classes which cover every possible method for making decisions. These are organized into two broad strategies as follows:

Strategy I: Policy Search. This strategy searches over parameterized functions to identify the ones that work best over time. These come in two classes:

1. **Policy function approximations (PFAs).** Analytical functions that map the information in the state variable direction to a decision. Some examples are:

    a. Buy low, sell high policies in finance – see asset selling in Topic 2.

    b. Order-up-to policies for inventories: If the inventory is below θ^{min} order up to θ^{max} and we then have to tune $\theta = (\theta^{min}, \theta^{max})$ – see inventory planning in Topic 2.

    c. Linear decision rules are special problems where we can write a policy as:
    $$X^{\pi}(S_t|\theta) = \sum_{g \in G} \theta_g \phi_g(S_t).$$

    d. A PFA can be any lookup table, parametric function (linear or nonlinear including neural networks) or nonparametric function.

    We note that PFAs include every possible functional form that we might use in machine learning.

2. **Cost function approximations (CFAs).** A parameterized optimization problem that is typically a deterministic approximation, in which parameters have been introduced to make it work well under uncertainty (see [7] for an introduction). CFAs are widely used in industry in an ad-hoc way, but I have not been able to find this strategy formally studied in the research literature. Some examples include:

    a. Solving the shortest path over a network with random link times, but use the θ-percentile of the travel times (instead of the mean) – see Dynamic shortest paths in Topic 5.

    b. Scheduling aircraft using an integer program, while inserting slack in the schedule to account for weather delays.

    c. Scheduling nurses but limiting their time to 32 hours per week to provide slack in case emergencies arise.

Strategy II: Lookahead approximations. These policies identify good decisions by optimizing across the current cost or contribution plus an approximation of the effect of a decision now on the future. Again, these come in two classes:

3. **Value function approximations (VFAs).** Policies based on VFAs cover all methods based on Bellman's equation, which approximates the downstream value of landing in a state. This approach has attracted tremendous attention under names such as approximate dynamic programming, adaptive dynamic programming, neurodynamic programming, and most commonly today, reinforcement learning. We illustrate Bellman's equation in Static shortest paths (Topic 5, section 5.1), but here the value functions are exact, since we take advantage that our "state" variable is simply which node where the traveler is located.

   We can formulate virtually any sequential decision problem using Bellman's equation, but the vast majority cannot be solved exactly. There is by now a substantial literature for estimating value functions approximately, although this approach is more popular in the academic literature rather than used in practice. An in-depth investigation of this strategy is beyond the scope of this course.

4. **Direct lookahead approximations (DLAs).** This is where we explicitly plan into the future to help make a decision now. DLAs can be split into two subclasses:

   a. Deterministic DLAs are when we ignore uncertainty to create a deterministic lookahead model, a strategy that is often called a rolling (or receding) horizon procedure, or model predictive control. We do this in Dynamic Shortest Paths (section 5.2).

      It is possible to parameterize the lookahead to help make it more robust to uncertainty, producing a hybrid CFA/DLA, as we do in section 5.2.

   b. Stochastic DLAs create an approximate stochastic lookahead model, typically using sampled approximations of random outcomes. This covers stochastic programming (with scenario trees), robust optimization and approximate dynamic programming, which can be used to solve a simplified stochastic lookahead. This policy is beyond the scope of this course.

These four (meta)classes of policies are universal – they include any method proposed in the research literature or used in practice. None of these methods is a panacea – depending on the specific characteristics of a problem, any one of these may work best. However, some are more useful than others. If we divide DLAs into two classes (deterministic lookahead and stochastic lookahead), we have five types of policies. These can be organized into three categories, ranging from the most to least widely used:

- Category 1: PFAs, CFAs and deterministic DLAs – This category is absolutely the most widely used. The choice of PFA, CFA and deterministic DLA tends to be obvious from the application.

- Category 2: Stochastic DLAs – There is a handful of problems where we need to plan into the future, and where we have to recognize that the future is uncertain.

- Category 3: Policies based on VFAs – Value function approximations are incredibly popular in the academic literature, but the number of applications in practice for VFAs is quite small.

This course focuses primarily on the policies in Category 1. These are by far the most widely used, and therefore are most appropriate for an introductory course on optimization. Categories 2 and 3 are useful for very special classes of problems, but are beyond the scope of an introductory course.

6.4 Evaluating Policies

The most common way to evaluate a policy is to simulate its performance using historical data as we did with the asset selling example, or simulated data as we did for inventory planning (see equation (6.17)). If we have access to multiple samples of the information process $W_1, \ldots, W_T$

which we represented using $W_t(\omega)$ for $\omega = \{\omega^1, \ldots, \omega^N\}$, we can use the average performance in equation (6.19). Both versions sum the contributions over the time periods $t = 0, 1, \ldots, T$.

There are times when we are running a series of experiments as we did in Topic 4 for finding the best treatment. Now imagine that we are trying to find the best combinations of chemicals to produce a new material, or we are testing different processes to create a new drug that we are evaluating in a lab. Alternatively, we might be using a simulator in a computer to test different sizes of a fleet of

These are settings where we do not care how well we do along the way – instead we just care how well we do at the end.

There are two objective functions we can use to evaluate a policy:

- Optimize the cumulative cost or contribution – Here we add up costs (or contributions) to evaluate the policy over time (or experiments). I like to call this the "cumulative reward" objective.

- Optimize the final cost or contribution – Here we run a series of experiments where we learn from the experiments, but we are not concerned with how well we do. Then, after using up our budget for learning, we have to make a final decision of what is the best choice, and then evaluate the performance of this choice. I call this the "final reward" objective.

A second issue we have to recognize is that we often have to distinguish between how well we expect to perform, and the risk that the performance might trigger a red flag that we would like to avoid. For example:

- In an inventory problem, we may want to minimize average costs (where we have a cost for lost demand), but where we want to make sure we cover at least 97 percent of demands.

- In a financial problem, we may want to place special emphasis on avoiding loses beyond some acceptable amount when we sell our asset.

We have seen an example of risk in our dynamic shortest path problem where we need to minimize the risk of arriving late for an appointment. This is handled in equation (5.3) where we add a penalty for late arrivals.

Risk is a very rich issue, but is beyond what should be covered in an introductory course.

Topic 7: Linear Programming

We finally get to linear programming. Unlike traditional optimization courses that might start with linear programming, we recognize that these are complex problems that only arise in very specialized situations. By this point in the course, students have seen optimization problems that everyone encounters. Now we are going to move into an important class of resource allocation problems that are much higher dimensional. These problems are important and arise in many business settings, but it is unlikely that students will have experienced these problems.

Linear programming is initially presented as a static problem, where we formulate a problem as a linear program, solve it, get the optimal solution, and then implement it (in some way). Virtually all of the original motivating applications for linear programs are, in fact, sequential decision problems, which means the linear program is actually a policy where the decisions are implemented over time, almost always in the presence of some form of uncertainty. We address this perspective after we treat the static problem.

We are going to progress in three steps:

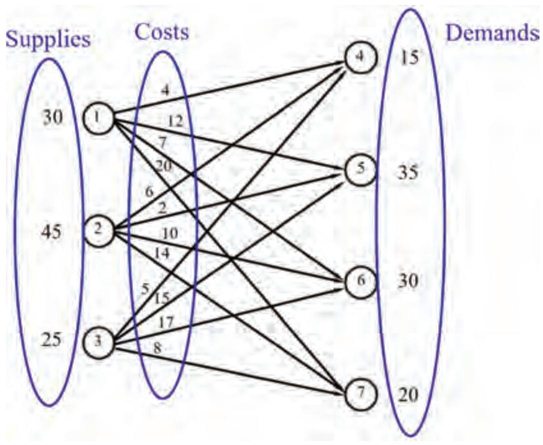
- Section 7.1 – We are going to use a basic resource allocation problem to illustrate a linear program. This will be the first time that we address a decision that is in the form of a vector. We are going to illustrate the simplex algorithm using a purely graphical approach (networks make this easy). I would note that I do not think it is necessary to teach the simplex algorithm, but it is popular material, and it does help in understanding dual variables.

- Section 7.2 – In this section I repeat the simplex algorithm but this time I show how to perform each step using linear algebra. I consider this material completely optional, but for faculty who enjoy presenting the simplex algorithm, the network problem makes it very easy to walk through the steps without having to resort to a two-dimensional linear program.

- Section 7.3 – Now we show that our so-called static linear program is a sequential decision problem, which means that our original LP is actually a policy for a sequential decision problem. This exactly parallels the transition from a static shortest path problem (section 5.1) to a dynamic shortest path problem (section 5.2).

7.1 As a Static Problem – The Simplex Algorithm I

There are many ways to illustrate the need for a linear program – one is the network problem below where we have supplies of resources at three locations, and we need to satisfy demands at four locations. Finding the optimal way to distribute these supplies to meet the demands can be solved as a linear program. In this section we illustrate the simplex algorithm applied to networks (this is known as "network simplex") where the entire presentation is graphical – no linear algebra. In section 7.2 I repeat the steps, but this time I include the linear algebra that goes with each step.



Using what we learned in Topic 6, we can write this out as a linear program using our canonical model

$$Ax = b, \tag{7.1}$$

$$x \leq u, \tag{7.2}$$

$$x \geq 0. \tag{7.3}$$

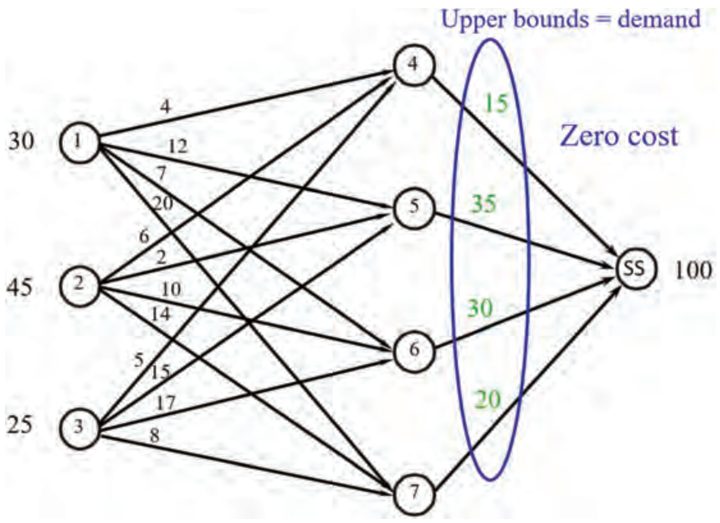
Writing out the constraints gives us

$$\sum_j x_{ij} = R_i \qquad \textit{for } i = 1, 2, 3. \tag{7.4}$$

$$\sum_i x_{ij} \le D_j \qquad \textit{for } j = 4, 5, 6, 7. \tag{7.5}$$

$$x_{ij} \ge 0 \qquad \textit{for } i = 1, 2, 3 \textit{ and } j = 4, 5, 6, 7. \tag{7.6}$$

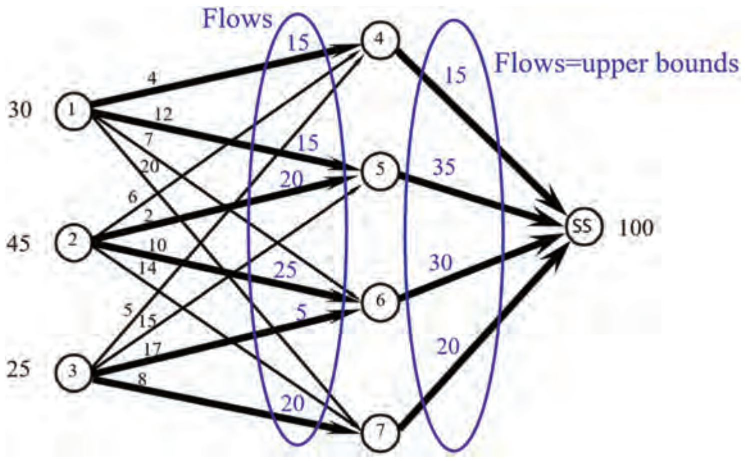
If you want to teach the simplex method, a nice way is to use the network above and show the steps of simplex graphically, rather than the usual treatment using matrices. We are going to start by having all the flows exit through a super sink with zero-cost links moving from each destination node (nodes 4–7) with upper bounds equal to the demand at the node:



The supersink, while not necessary for the algorithm, will help to simplify the presentation.

We first need an initial feasible solution, and this solution has to represent what is known as a *basis*. A basis (for a network problem) is a set of links with flows that satisfy all flow conservation constraints, along with all upper and lower bounds.

There are different ways to get an initial feasible solution. For this network problem, we can start by putting the required flow into each destination node (nodes 4–7) on the link into the supersink. Then, we start at node 4, find a link from a supply node (node 1), and put as much flow on this link as we can. Since the demand at node 4 is only 15, we cannot put more than 15. Also, since the supply at node 1 is 30, we have to take the smaller of 30 and 15 and put this amount (15) on the link (1–4).



Now we go to node 1 where we still have 15 units of unassigned flow. Node 5 needs 35 units of flow, but we only have 15 remaining, so we put 15. Next we go to node 5, where we still have an unsatisfied demand of 20, and look to the next supply node, node 2, which has 45 units of available flow. We take 20 of these to put on the link (2–5) which now gives us our required flow of 35 into node 5.

Next we move to node 2 where we still have 25 unassigned units of flow. We turn to node 6 that needs 30 units of flow, and push all 25 units of flow from 2 to 5. We still need 5 more units of flow at node 6, so we move down to node 3 and take 5 out of the available 25 units of flow.

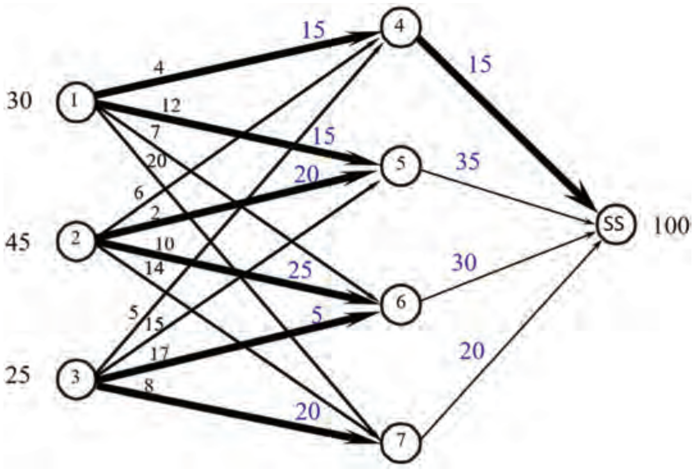
Finally, we move the remaining 20 units of flow at node 3 to node 7.

We are not quite done. To be a basis (for a network), the set of links in the basis must satisfy two conditions:

1. All links with flow strictly between the upper and lower bound must be in the basis.

2. The set of links in the basis must form of tree.

So, we quickly see our links fail condition 2 – the set of links with flow do not form a tree. But we are not required to keep links with flow if the flow is at the upper or lower bound. Of the four links into the supersink, only one can be in the basis, so we are going to arbitrarily choose the first one, which gives us the basis:

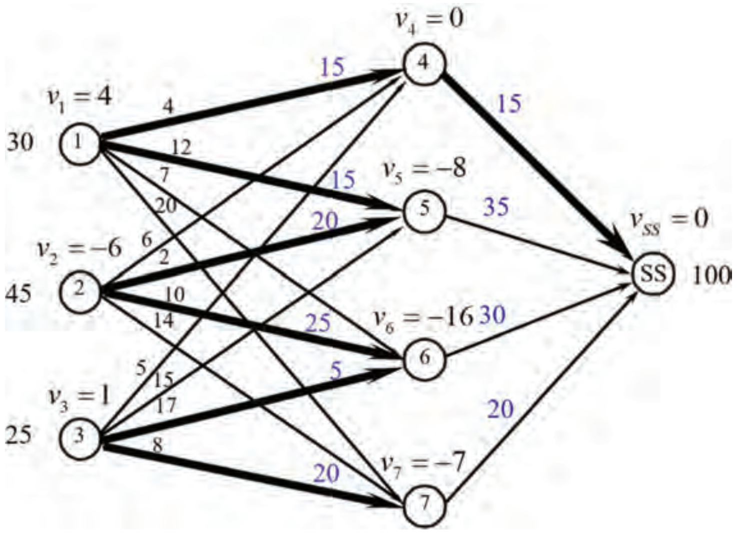


We are not claiming that this is optimal (it is not), but we now have a way of finding the optimal solution. The first step is to compute a "value" (known as a "dual variable") which is the cost of moving a unit of flow from each node to the supersink. Remember that all the links directly attached to the supersink have 0 cost. So, the value at node 4 would be 0, since this is the cost of the path from 4 to SS.

To get from node 1 to SS, we have to move 1–4 (cost 4) and then 4-ss (cost 0) which is a cost of 4. A better way to get the dual at node 1 is to see that the path moves from 1 to 4 (at a cost of 4), and then just add the dual at 4 (which is 0).

To get from node 5 to SS, we first have to go backwards on the link (1–5), so this is a cost of minus 12. The dual at 1 is 4, so the dual at 5 is $-12 + 4 = -8$.

The dual at node 2 would be the cost of 2 to get 2–5, plus the dual at 5 $= -8$, so the dual at $2 = 2 + (-8) = -6$. Continuing this logic for the remaining nodes gives us the duals:



You can quickly check that each dual v_i is the cost from that node to the supersink, following links that are in the basis.

Note at the same time there is always a single path from each node to the basis. This is a key property of the basis, which we are about to exploit.

Now let's optimize. We are going to do this by looking at the nonbasic links without flow, and ask: What is the value of increasing flow on a nonbasic link?

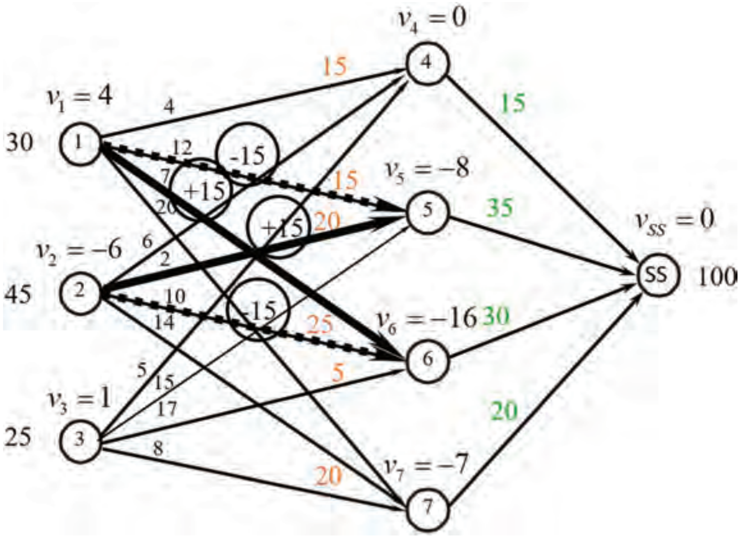
Let's take the link (1–6). To move one more unit of flow from 1 to 6, we are going to first add the flow from 1–6 (at a cost of 7), then we are going to move one unit of flow from 6 to the super sink, at a cost $v_6 = -16$ (the dual variable for node 6), and then we are going to move one unit of flow from the super sink back to node 1. The cost of moving flow from SS to node 1 is negative the dual for node 1 (since this is the cost to go from 1 to SS). This means moving a unit of flow from 1 to 6, then 6 to SS, and finally SS back to 1, is

$$\bar{c}_{16} = c_{16} + v_6 - v_1.$$

The cost $\bar{c}_{16}$ is called the *reduced cost* since the cost is "reduced" by the changes needed to guarantee that we still satisfy all the constraints. This is a very simple calculation, which means we can do this calculation for every link that is not in the basis (note that the reduced cost for any link in the basis is equal to zero). For link 1–6, the reduced cost is

$$\bar{c}_{16} = c_{16} + v_6 - v_1 = 7 + (-16) - 4 = -13.$$

The reduced cost captures the change in all the costs if we add one more unit of flow from 1 to 6, and then make all the other adjustments needed to ensure that we are still satisfying all the other constraints. Since the reduced cost is negative, this means that for each unit of flow, total costs will go down by 13, which means we get a better solution.



Now we just have to figure out how much flow we can move. We first note that adding one unit of flow from 6 to SS, and then moving one unit of flow from SS to 1, means there is no change on links 1–4 and 4-SS. The only links where flow actually changes are the links on the path along the links in the basis from 6 back to 1. This means flow increases on the links 1–6 and 2–5, while flow decreases on links 1–5 and 2–6. We want to move flow until the first link hits its lower bound (or upper bound if we had these, but we don't). The link 1–5 has 15 units of flow, while link 2–6 has 25 units of flow, so link 1–5 will be the first link to hit zero. This means we can move 15 units of flow, at which point we would stop and drop link 1–5 from the basis (since it no longer has flow), while we now add link 1–6 to the basis.

We now have a new basis which means we have to update all our dual variables v_i. There are ways to do this very quickly, but these are technical issues that are not important for our discussion. Remember – we are not trying to teach students how to code the simplex algorithm – we are trying to teach important concepts in optimization.

With the new dual variables, we now have to recompute the reduced costs for all the nonbasic links (note that linear programming packages have tricks to do this very quickly). If we find another link with a negative reduced cost, then we have to repeat this exercise. We keep doing this until we no longer find any links with a negative reduced cost. At this point, we have found the optimal solution.

It is possible that when we route flow around the cycle, two links may hit zero at the same time. If this happens, we drop only one from the basis, and leave the other link with zero flow in the basis. This is known as a *degenerate basis.* A byproduct of a degenerate basis is that it is possible that we may find that the amount of flow we can move around the cycle (when we find a nonbasic link with negative reduced cost) is zero. Nothing wrong with this – it is actually fairly common.

This is a peek into the simplex method for network problems. The simplex algorithm works for linear programs that are not networks, and in this case we cannot draw these pretty pictures. But the basic idea is the same. Modern implementations of the simplex algorithm involve a vast array of engineering tricks to make the algorithm extremely fast. What is most important for students to understand is that we have exceptionally fast algorithms for solving linear programs, and free software is widely available.

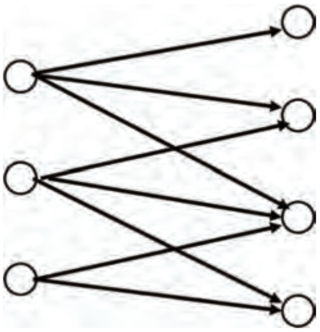
Next, we are going to illustrate a dynamic inventory problem where we repeatedly solve linear programs over time.

7.2 The Simplex Algorithm II – with the Matrix Linear Algebra

For a first course on optimization, I do not think it is necessary to show the matrix linear algebra behind the simplex algorithm, but there will be instructors who will want to include this material. This section includes all the slides from one of my lectures that presents the network simplex algorithm alongside the matrix calculations. You can download the PowerPoint slides that contain this material from https://tinyurl.com/PowellNetworkSimplex/.

The illustration below uses the network in section 7.1, but without the supersink. We choose node 1 arbitrarily to be the root node (and we explain why we need the concept of a "root node" for networks in the discussion below).

A generic linear programming model for a network problem is written as follows:



$$\min_x \sum_{supplies\ i} \sum_{demands\ j} c_{ij} x_{ij}$$

subject to the constraints:

$$\sum_j x_{ij} = S_i = \text{Supply at node i}$$
$$\sum_i x_{ij} = D_j = \text{Demand at node j}$$
$$x_{ij} \geq 0$$

We can write these constraints in matrix form as

$$Ax = b,$$

$$x \geq 0.$$

Now assume we are going to solve the numerical example below. Our constraints would look like

$$x_{14} + x_{15} + x_{16} = 12$$

$$x_{24} + x_{25} + x_{26} + x_{27} = 18$$

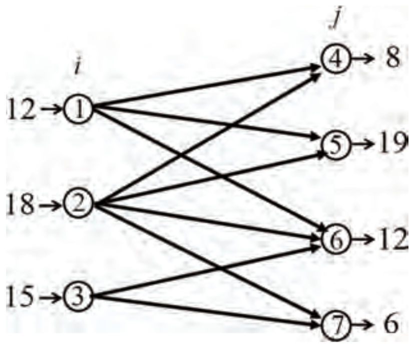
$$x_{36} + x_{37} = 15$$

$$-x_{14} - x_{24} = -8$$

$$-x_{15} - x_{25} = -19$$

$$-x_{16} - x_{26} - x_{36} = -12$$

$$-x_{27} - x_{37} = -6$$



$$x_{14}, x_{15}, x_{16}, x_{24}, x_{25}, x_{26}, x_{27}, x_{36}, x_{37} \geq 0$$

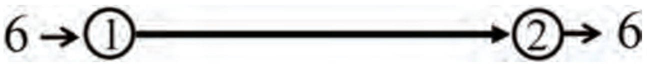
We would write our constraint matrix A with a row for each constraint, and a column for each variable x_{ij}, giving us the ematrix:

$$\text{Nodes}\downarrow \quad \xleftarrow{\quad} \text{Links} \xrightarrow{\quad}$$

$$A = \begin{array}{c|ccccccccc} & 1-4 & 1-5 & 1-6 & 2-4 & 2-5 & 2-6 & 2-7 & 3-6 & 3-7 \\ \hline 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & 0 \\ 7 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & 0 & -1 \end{array}$$

Some notes:

- If we add the rows of the matrix A, they sum to zero. This means that one of the rows is redundant.... If we drop one of the rows, all the constraints will still be satisfied. In other words, if we have a network with n nodes, we need $n - 1$ constraints.
- The simplest way to illustrate this property of networks is to consider a network with two nodes and one link:



- If we enforce flow conservation at node 1, this means we will be sending 6 units of flow from 1 to 2, which automatically satisfies the flow conservation constraint at node 2.
- So, for larger networks, we can pick any node and drop its constraint. This node is called the *root node.* If r is the root node,

then the dual variable $v_r = 0$ (remember – this is the cost of moving a unit of flow from that node to the root node).

- For our network above, we can arbitrarily pick node 1 as the root node. This gives us the following constraint matrix:

$$A = \begin{array}{c} \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1-4 & 1-5 & 1-6 & 2-4 & 2-5 & 2-6 & 2-7 & 3-6 & 3-7 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \end{array}\right] \end{array}$$

The simplex algorithm for linear programming requires that we start with a *basis* which is a set of variables x_{ij} which we are going to adjust to guarantee that the constraints are satisfied any time we change a variable that is not in the basis. Our only requirement for a nonbasic variable is that it must be at its lower or upper bound. In our numerical example (without the supersink), we do not have any variables with upper bounds, so all nonbasic variables must equal 0.

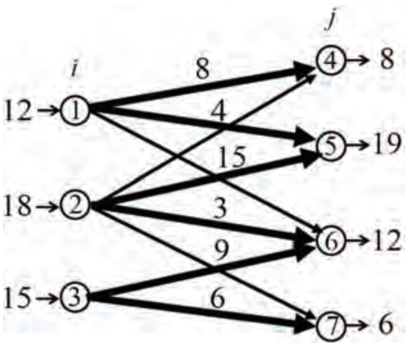
These concepts are best explained by example.

We have to start by creating our basis. We need to find a set of flows that satisifes the constraints. We do not care about the quality of the solution – our simplex algorithm can start with any feasible solution that satisfies the rules for our basic and nonbasic variables.

Professional linear programming packages have sophisticated logic for creating starting solutions. For our simple problem, we are going to use a simple strategy called the "northwest corner rule" where we literally start in the northwest corner of our graph (that is, node 1), and start assigning flow to the northeast corner (which would be node 4). We assign as much as we can (that is, the smaller of either the supply at node 1 or the demand at node 4). We then move to either the next supply node (if we allocated all the supply from node 1) or the next demand node (if we satisfied all the demand at node 4), and keep repeating the process using the remaining nodes with unused supply or unsatisfied demand.

This process produces the network below, along with the basic vector x^B (the vector of links x_{ij} for links in the basis) and the nonbasic vector x^N (the links not in the basis):

$$x^B = \begin{bmatrix} 1-4 \\ 1-5 \\ 2-5 \\ 2-6 \\ 3-6 \\ 3-7 \end{bmatrix} \qquad x^N = \begin{bmatrix} 1-6 \\ 2-4 \\ 2-7 \end{bmatrix}$$



We can now partition our A-matrix into a square matrix A^B of the basic variables (it will always be square) and the remaining matrix A^N comprised of the non-basic columns:

$$A^B = \begin{array}{c|cccccc|} & 1-4 & 1-5 & 2-5 & 2-6 & 3-6 & 3-7 \\ 2 & 0 & 0 & 1 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 & 1 & 1 \\ 4 & -1 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & -1 & -1 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & -1 & -1 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}$$

$$A^N = \begin{array}{c|ccc|} & 1-6 & 2-4 & 2-7 \\ 2 & 0 & 1 & 1 \\ 3 & 0 & 0 & 0 \\ 4 & 0 & -1 & 0 \\ 5 & 0 & 0 & 0 \\ 6 & -1 & 0 & 0 \\ 7 & 0 & 0 & -1 \end{array}$$

We can now restate our constraints using the vectors of basic and nonbasic variables, and the corresponding columns of the $A-$ matrix. The constraints

$$Ax = b$$

becomes

$$[A^B\ A^N]\begin{bmatrix}x^B\\ x^N\end{bmatrix}=b$$

which is the same as

$$A^Bx^B+A^Nx^N=b.$$

We can now solve for thebasic variables x^B in terms of the basic variables x^N:

$$x^B=[A^B]^{-1}[b-A^Nx^N].$$

For our network problem, we have no upper bounds so $x^N=0$, but this will not always be the case. Note that we can guarantee that the matrix A^B is, in fact, invertible by how we have constructed the basis.

For general linear programming problems we need some reasonably sophisticated linear algebra to handle the matrix inversion $[A^B]^{-1}$. Remember that linear programming models have been solved with millions of variables. In fact, network problems (which have a lot of structure) can be solved even with tens of millions of variables. We would not even be able to store a matrix A^B with millions of rows and columns. This is where specialists use a lot of tricks.

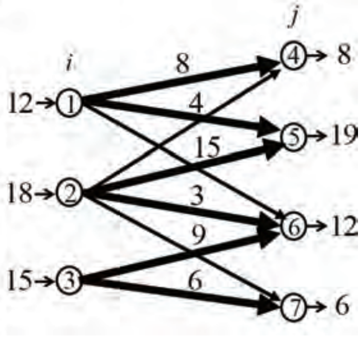
In fact, we are going to show you how you can invert the basis matrix A^B for our network problem *by inspection!*

Recall that each row of the basis matrix A^B corresponds to a node, while each column corresponds to a link (that is, a decision variable x_{ij}). This is why we often call A^B a "node-arc incidence matrix." It turns out that each row of the inverse $[A^B]^{-1}$ corresponds to a link, while each column corresponds to a path from a node to the root node. The element of the matrix $[A^B]^{-1}$ indicates if the link for that row is in the path from the node for that column. We use 0 if the link is not in the path, and then $+1$ or -1 to indicate if the link is in the path, and whether you have to traverse the link in the forward direction or backwards. The choice of whether it is $+1$ or -1 depends on what sign convention you have used in your flow conservation constraints (that is, did you write flow out minus flow in, or the reverse). Below is the inverse that we get for our problem.

To verify that $[A^B]^{-1}$ is in fact the inverse of A^B, we can perform the multiplication $A^B[A^B]^{-1}$ to verify that we get the identity matrix.

Links ↓ ⟵ Paths ⟶

$$\left[A^B\right]^{-1} = \begin{array}{c|cccccc} & 2-1 & 3-1 & 4-1 & 5-1 & 6-1 & 7-1 \\ \hline 1-4 & 0 & 0 & -1 & 0 & 0 & 0 \\ 1-5 & -1 & -1 & 0 & -1 & -1 & -1 \\ 2-5 & 1 & 1 & 0 & 0 & 1 & 1 \\ 2-6 & 0 & -1 & 0 & 0 & -1 & -1 \\ 3-6 & 0 & 1 & 0 & 0 & 0 & 1 \\ 3-7 & 0 & 0 & 0 & 0 & 0 & -1 \end{array}$$



This is done below:

$$\overset{A^B}{\begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}} \times \overset{[A^B]^{-1}}{\begin{bmatrix} 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & -1 & 0 & -1 & -1 & -1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & -1 & 0 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}}$$

$$= \overset{I}{\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}}$$

Note that if we perform this multiplication and find we get $-I$ then you just have to switch your sign convention. Obviously you only have to check this for one element.

We can rewrite our objective function using

$$\operatorname*{Min}_x c^T x = \left(c^B\right)^T x^B + \left(c^N\right)^T x^N$$

$$
\begin{aligned}
&= (c^B)^T([A^B\}^{-1}(b - A^N x^N)) + (c^N)^T x^N \\
&= (c^B)^T [A^b]^{-1} b - (c^B)^T [A^B]^{-1} A^N x^N + (c^N)^T x^N \\
&= (c^B)^T [A^B]^{-1} b + \overline{c}^N x^N
\end{aligned}
$$

where $\overline{c}^N$ is the vector of "reduced costs" for the nonbasic links given by

$$\overline{c}^N = (c^N)^T - (c^B)^T [A^B]^{-1} A^N$$

Reduced costs tell us if we should increase the flow on a nonbasic link, while adjusting flows on all the basic links so that the flow conservation (plus upper and lower bound) constraints are satisfied. For our numerical example above, the reduced costs are calculated as follows:

$$\overline{c}^N = [c_{16} \quad c_{24} \quad c_{27}] - [c_{14} \quad c_{15} \quad c_{25} \quad c_{26} \quad c_{36} \quad c_{37}]$$

$$
\begin{array}{c}
\text{“paths”} \\
[\,2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7\,] \\
\begin{bmatrix}
0 & 0 & -1 & 0 & 0 & 0 \\
-1 & -1 & 0 & -1 & -1 & -1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
0 & -1 & 0 & 0 & -1 & -1 \\
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
\end{array}
\begin{array}{c}
\text{Nonbasic} \\
\text{links} \\
\begin{array}{ccc} 1-6 & 2-4 & 2-7 \end{array} \\
\begin{bmatrix}
0 & 1 & 1 \\
0 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 0 \\
-1 & 0 & 0 \\
0 & 0 & -1
\end{bmatrix}
\end{array}
$$

Continuing the calculations:

$$\overline{c}^N = [c_{16} \quad c_{24} \quad c_{27}] - [c_{14} \quad c_{15} \quad c_{25} \quad c_{26} \quad c_{36} \quad c_{37}]$$

$$
\begin{bmatrix}
0 & 0 & -1 & 0 & 0 & 0 \\
-1 & -1 & 0 & -1 & -1 & -1 \\
1 & 1 & 0 & 0 & 1 & 1 \\
0 & -1 & 0 & 0 & -1 & -1 \\
0 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & -1
\end{bmatrix}
\begin{bmatrix}
0 & 1 & 1 \\
0 & 0 & 0 \\
0 & -1 & 0 \\
0 & 0 & 0 \\
-1 & 0 & 0 \\
0 & 0 & -1
\end{bmatrix}
$$

$$= [c_{16} \quad c_{24} \quad c_{27}] - \overbrace{[v_2 \quad v_3 \quad v_4 \quad v_5 \quad v_6 \quad v_7]}^{(c^B)^T[A^B]^{-1}} \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix}$$

$$= [c_{16} \quad c_{24} \quad c_{27}] - [-v_6 \quad v_2 - v_4 \quad v_2 - v_7]$$

$$= [c_{16} \quad - v_1 + v_6 \quad c_{24} - v_2 + v_4 \quad c_{27} - v_2 + v_7]$$

We see that $(c^B)^T[A^B]^{-1}$ is the inner product of the link costs (for basic links) times the link-path incidence matrix $[A^B\}^{-1}$. This means that $(c^B)^T[A^B]^{-1}$ is the vector of dual variables, which as we have seen are the path costs along the basis from each node to the root node. All of this reduces to the simple relationship (for networks) between costs and reduced costs for nonbasic links

$$\bar{c}^N = [c_{16} - v_1 + v_6 \quad c_{24} - v_2 + v_4 \quad c_{27} - v_2 + v_7].$$

We can compute the path costs (dual variables) v_i just by following the path along the basis from each node to the supersink (node 1). Remember we have to subtract the cost for any link that we traverse in the reverse direction of the link. This gives us

$$v_1 = 0$$

$$v_2 = 16 - 8 = 8$$

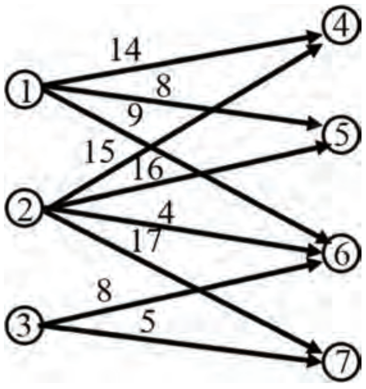
$$v_3 = 8 - 4 + 16 - 8 = 12$$

$$v_4 = -14$$

$$v_5 = -8$$

$$v_6 = -4 + 16 - 8 = 4$$

$$v_7 = -5 + 8 - 4 + 16 - 8 = 7$$



The reduced costs on the nonbasic links are then

$$\bar{c}^N = [\bar{c}_{16} \quad \bar{c}_{24} \quad \bar{c}_{27}]$$

$$\begin{aligned}
&= [c_{16} - v_1 + v_6 \quad c_{24} - v_2 + v_4 \quad c_{27} - v_2 + v_7] \\
&= [9 - 0 + 4 \quad 15 - 8 + (-14) \quad 17 - 8 + 7] \\
&= [13 \quad -7 \quad 16]
\end{aligned}$$

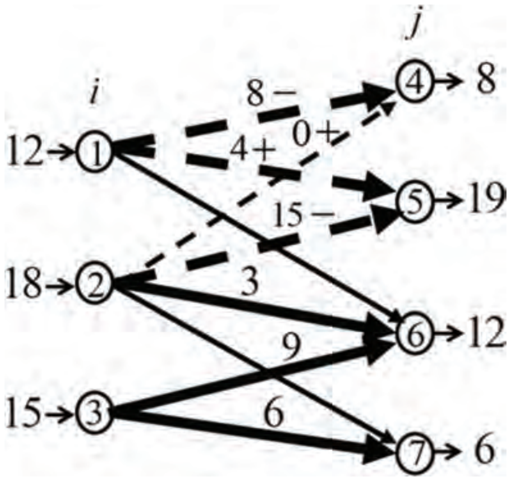
The reduced cost tells us how much total costs will change if we increase the flow on each nonbasic link. If the reduced cost is negative, then we have found a nonbasic link where we can increase the flow and reduce total costs. We now search for nonbasic links with a negative reduced cost, since this means increasing the flow on that link, and then adjusting the flows on the basic links so we maintain flow conservation, will reduce total costs.

There are different strategies for choosing the nonbasic link, since networks can be quite large (from many thousands of links to millions of links). It makes sense to choose the nonbasic link with the most negative reduced cost, but this would mean calculating all the reduced cost and finding the smallest. Computer scientists have refined these strategies to balance the time required to find the best nonbasic link that produces the fastest convergence.

From our list of reduced costs above, we see that the only nonbasic link with a negative reduced cost is link (2,4). We want to increase flow from 2 to 4. Then, to maintain flow conservation, for each unit of flow we push from 2 to 4, we want to push a unit from 4 to the root node (node 1), and then from the root node to node 2, always limiting ourselves to links in the basis. We note that there is always exactly one path between any node and the root node.

As we see the graph to the right, the path from 4 to the root node means reducing a unit of flow on the link from 1 to 4. Then, we have to push a unit of flow from node 1 to node 2, which means increasing flow on link (1,5) and then decreasing flow on link (2,5).

The next step is that we have to figure out how much flow to move. The answer to this is simple: we move as much as possible. We look at each link that is losing flow, and then calculate how much flow is on these links, and take the link with the smallest flow. If links have upper bounds, then we also look at each link that is gaining flow, and take the link that can increase flow by the smallest amount before hitting



the upper bound. Finally, we choose the link losing flow where we can lose the smallest amount of flow, and compare it to the links gaining flow, and choose the link that can gain the least. Finally, we choose the link that can lose (or gain) the smallest amount of flow, and that link determines how much flow we can lose. The constraining link is also the link that we drop from the basis.

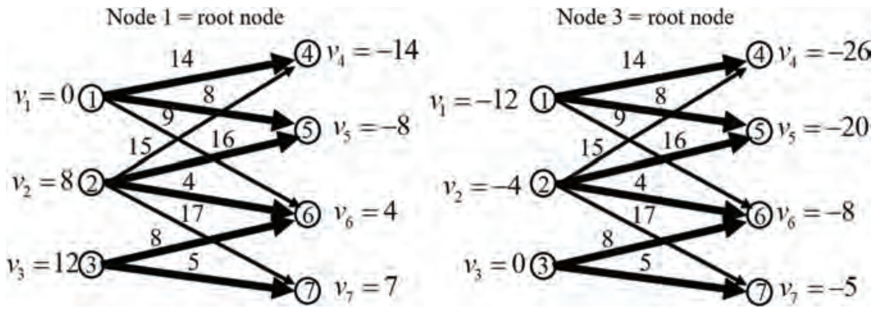
After we complete the process of adding the new nonbasic link and dropping the constraining link, we have to recompute the dual variables v_i. Again, we note that commercial software uses a variety of programming tricks to accelerate this process.

Some notes:

- It is possible that there may be a tie – two links losing flow have the same amount of flow, or the link that can lose the least flow matches the link that can gain the least flow. In case of ties, we just pick one link arbitrarily to drop from the basis. After adding the new nonbasic link, we regain a valid basis (that is, all the links in the basis always form of a tree).

- It is also possible that the amount of flow that we can move is zero. This can (and will) happen, and is known as a "degenerate pivot." Even when the amount of flow we can move is zero, we still go through the same process of adding the new nonbasic link to the network, and dropping one of the constraining links (if there is more than one).

We note that all of our calculations seem to depend on our choice of root node. It turns out that changing the root node has the effect of changing all of the dual variables by a constant, which means that the reduced costs, which all involve differences between dual variables, are not affected. Below are two networks with different root nodes to illustrate this property.



This nice property also hints at a limitation in the interpretation of a dual variable. It is common to think of a dual variable v_i as being the marginal value of the resources entering or leaving the network at node i. Since the sum of the supplies and demands for our network problem must sum to zero, it does not make sense to perturb the flow entering or leaving the network at node i, since we also have to specify the change in the flow at some other node so that the supplies and demands remain balanced. We have actually addressed this problem by dropping one of the constraints (the root node). This means that if we perturb the supply or demand at some node i, we are implicitly balancing this change by adding or subtracting the same amount of flow at our root node so that the sum of supplies and demands remain balanced.

7.3 As a Policy for a Dynamic Problem

Let's start with the same network problem we used above, but now assume that we are using this to match available inventories of products in distribution centers to the demands of retailers looking to restock their inventories. We might reasonably solve this problem daily (or weekly), but in this case we are now solving a sequential decision problem, where our "policy" requires solving the linear programming problem.

Recognizing that we are solving this problem each time period, our optimization problem at time t would be written

$$\min_{x_t} \sum_{i,j} c_{tij} x_{tij}$$

subject to

$$\sum_j x_{tij} = R_{ti} \quad for\ i = 1, 2, 3, \tag{7.7}$$

$$\sum_i x_{tij} \leq D_{tj} \quad for\ j = 4, 5, 6, 7, \tag{7.8}$$

$$x_{tij} \geq 0 \quad for\ i = 1, 2, 3\ and\ j = 4, 5, 6, 7, \tag{7.9}$$

where the supplies at time t are given by R_{ti} and the demands are given by D_{tj}. We have also allowed our costs c_t to be time-dependent. We quickly see that we can minimize costs by not satisfying any demand, so a better model might be to maximize profits. Let

p_{tj} = the price we receive for satisfying a unit of demand at node j.

Our objective function would then be

$$\max_{x_t} \left(\sum_j p_{tj} \left(\min \left\{ D_{tj}, \sum_i x_{tij} \right\} \right) - \sum_{i,j} c_{tij} x_{tij} \right) \tag{7.10}$$

A more realistic model needs to be modified to reflect the possibility that total supply may be greater than total demand, or less than total demand. We have already written the demand constraint (7.8) as an inequality. However, we need to add the option that allows us to hold excess inventory until time $t + 1$. In fact, we may even want to hold inventory while not satisfying demand. Remember that in our model we are allowing costs and prices to vary over time. There may be a time period where costs rise and/or prices drop, at which point we prefer to hold our inventory for a future time period when prices and costs may be more favorable.

We may face uncertainty in our available inventories (these might include inventories arriving soon, but perhaps they are delayed), or the demands (which may be higher or lower than expected). We might, then, replace constraints (7.7)–(7.9) with

$$\sum_j x_{tij} = \theta^R R_{ti} \quad for\ i = 1, 2, 3. \tag{7.11}$$

$$\sum_i x_{tij} \leq \theta^D D_{tj} \quad for\ j = 4, 5, 6, 7. \tag{7.12}$$

$$x_{tij} \geq 0 \quad for\ i = 1, 2, 3\ and\ j = 4, 5, 6, 7. \tag{7.13}$$

The objective function (7.10) with constraints (7.11)–(7.13) looks like another optimization problem, but there is an important difference. This is just a problem at time t, where we want to maximize profits over time, not just at a point in time. If we choose to maximize (7.10) to get our decisions of what to do at time t, then we would say that this problem is now a *policy* which should be written

$$X^\pi(S_t|\theta) = \underset{x_t}{\operatorname{argmax}} \left(\sum_j p_{tj} \left(\min \left\{ D_{tj}, \sum_i x_{tij} \right\} \right) - \sum_{i,j} c_{tij} x_{tij} \right) \tag{7.14}$$

which has to be solved subject to the constraints (7.11)–(7.13), where the coefficient θ^R and θ^D are chosen to improve the performance of the policy over time. Thus, we can use our simplex algorithm to find the optimal solution of our linear program, but solving (7.14) is *not an optimal policy!*

We might create a more sophisticated policy by optimizing into the future, just as we did in our dynamic shortest path problem where we would plan a path to the destination, which would then be updated as new information came in. Such a *direct lookahead (DLA)* policy could be written

$$X_t^\pi(S_t|\theta) = \underset{\tilde{x}_{tt},,\ldots,\tilde{x}_{t,t+H},}{\operatorname{argmax}} \left(\sum_{t'=t}^{t+H} \left(\sum_j \tilde{p}_{tt'j} \left(\min \left\{ \tilde{D}_{tt'j}, \sum_i \tilde{x}_{tt'ij} \right\} \right) \right.\right.$$
$$\left.\left. - \sum_{i,j} \tilde{c}_{tt'ij} \tilde{x}_{tt'ij} \right) \right) \tag{7.15}$$

subject to, for $t' = t, t+1, \ldots t+H$:

$$\sum_j \tilde{x}_{tt'ij} = \theta^R \tilde{R}_{t'ti} \qquad \mathit{for}\ i = 1,2,3, \tag{7.16}$$

$$\sum_i \tilde{x}_{tt'ij} \leq \theta^D \tilde{D}_{tt'j} \qquad \mathit{for}\ j = 4,5,6,7, \tag{7.17}$$

$$\tilde{x}_{tt'ij} \geq 0 \qquad \mathit{for}\ i = 1,2,3\ \mathit{and}\ j = 4,5,6,7. \tag{7.18}$$

Note that we have put tilde's on all the variables used to plan into the future. Also, each of the tilde-variables has two time indices: time t, since we are solving the problem at time t, and time t', which is the time we are planning into the future. Other variables such as prices or demands are *forecasts* of future prices or demands made at time t.

Note that we do not care about the entire optimal solution over the horizon $t, \ldots, t+H$. We are only going to implement the first time period, which means we would set

$$x_t = \tilde{x}_{tt}.$$

Note that our DLA policy in (7.15)–(7.18) is still not an optimal policy, but it might be quite good. One challenge is that we have to tune the parameters

$$\theta = (\theta^R, \theta^D).$$

We need to first write out how we would simulate our policy $X_t^\pi(S_t|\theta)$. This requires identifying how random events (e.g. random demands, travel times, ...) affect how the solution x_t behaves in practice. This is a key step that is almost always overlooked when modeling and solving linear programs. What you have to do is to imagine that you are simulating the process. These equations represent the transition function in our sequential decision model which we represent compactly as

$$S_{t+1} = S^M(S_t, x_t, W_{t+1}).$$

Assume as we have done before that we can generate a series of sample paths where the n^{th} sample path is

$$\omega^n = (W_1^n, W_2^n, \ldots, W_T^n).$$

Let S_t^n be the state at time t while we are following sample path ω^n. Specifying ω^n refers to a specific sample of anything random (demands, prices, travel times, …). The value of the policy for sample path ω^n might be written

$$F^n(\theta) = \sum_{t=0}^{T} C(S_t(\omega^n), X_t^\pi(S_t(\omega^n)|\theta)),$$

where $X_t^\pi(S_t(\omega^n)|\theta)$ is the policy computed from (7.15)–(7.18). If we generate $n = 1, \ldots N$ sample paths, we can evaluate the policy by taking an average

$$\overline{F}^\pi(\theta) = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{T} C(S_t(\omega^n), X_t^\pi(S_t(\omega^n)|\theta)).$$

There are, of course, different ways to parameterize the policy. This means that when we optimize over policies, it requires designing different parameterizations, and then tuning each one. Let $f \in F$ be a family of parameterizations (we have to make these up by hand), and let $\theta \in \Theta^f$ be the parameters for parameterization f. Our policy $\pi = (f, \theta)$ consists of the parameterization and its tunable parameters.

We can now write out our optimization problem as

$$\min_{\pi=(f,\theta)} \overline{F}^\pi(\theta).$$

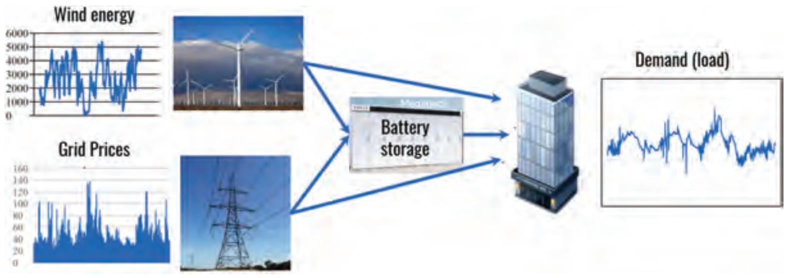
This discussion has illustrated that we are going to need to identify and compare different types of policies (such as (7.15)–(7.18)) as well as tuning any parameters that we have inserted. Coming up with different types of policies parallels choosing between the linear and nonlinear models we illustrated in Topic 1 on machine learning. Tuning the parameters for the policies parallels fitting our parametric models in machine learning.

While this process may seem ad hoc, it is no more ad hoc than searching among different statistical models in machine learning. Furthermore, this is precisely how the vast majority of sequential decision problems are solved in practice.

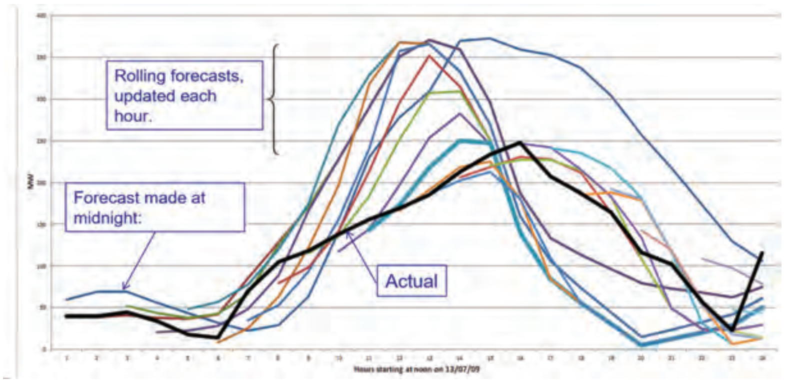
Topic 8: Dynamic Inventory Problems – Energy Storage

Readings: SDAM Chapter 9.

Here we address a more complex energy storage problem:



We have to decide how much to draw from a windfarm (with variable supply), the grid (with variable prices), to meet a predictable load (demand) for a building, using an energy storage device to absorb variations. We have rolling forecasts of wind which change quite a bit from hour to hour:



We are going to solve this much as we solve our dynamic shortest path problem, where we look into the future and pretend the various forecasts (such as wind) are perfectly accurate. This "lookahead model" is another example of a linear program. Even though the decision at each point in time is a scalar, we have to optimize the decisions over the entire horizon, which means we have to optimize over the entire vector of decisions.

Now, to handle uncertainty, we insert a coefficient $\theta_{t'-t}$ for our forecast of wind made at time t, of what the energy from the wind farm will be at time t'. Note that $\theta_{t'-t}$ is not a function of t, it is just a function of the difference $t'-t$. So if we look 24 hours into the future, we would have 24 coefficients. A simpler strategy to get us started would be to assume that there is just one coefficient for all forecasts. The modified lookahead linear program looks like:

$$X_t^{D-LA}(S_t|\theta) = \underset{x_t,(\tilde{x}_{tt'},t'=t+1,\ldots,t+H)}{\arg\min}\left(C(S_t,x_t)+\left[\sum_{t'=t+1}^{t+H}\tilde{c}_{tt'}\tilde{x}_{tt'}\right]\right)$$

$$\begin{aligned}
\tilde{x}_{tt'}^{wd} + \beta\tilde{x}_{tt'}^{rd} + \tilde{x}_{tt'}^{gd} &\le f_{tt'}^{D}\\
\tilde{x}_{tt'}^{gd} + \tilde{x}_{tt'}^{gr} &\le f_{tt'}^{G}\\
\tilde{x}_{tt'}^{rd} + \tilde{x}_{tt'}^{rg} &\le \tilde{R}_{tt'}\\
\tilde{x}_{tt'}^{wr} + \tilde{x}_{tt'}^{gr} &\le R^{\max}{}_{tt'}\\
\boxed{\tilde{x}_{tt'}^{wr} + \tilde{x}_{tt'}^{wd} \le \theta_{t'-t} f_{tt'}^{E}}\\
\tilde{x}_{tt'}^{wr} + \tilde{x}_{tt'}^{gr} &\le \gamma^{charge}\\
\tilde{x}_{tt'}^{rd} + \tilde{x}_{tt'}^{rg} &\le \gamma^{discharge}
\end{aligned}$$

This closely parallels the idea of solving a shortest path with modified costs. We solved that problem using our shortest path algorithm. This time, we need the full power of a linear program that we introduced in Topic 7.

Our next challenge is to tune the parameter vector θ which could be a scalar (if we use one parameter for all the forecasts) or, in our case, a 24-dimensional vector (one for each hour into the future). We first need to write out how we are going to evaluate our policy. If $X_t^{D-LA}(S_t|\theta)$ is our policy above, assume that it returns a decision vector x_t that determines what to do right now.

Let $C(S_t, x_t)$ be our performance metric (e.g. total costs) that occur just at time t. Now let $W_1, W_2, \ldots, W_t, \ldots, W_T$ be a sample realization of all the new information arriving, where W_t is the information that arrives between $t-1$ and t. This would include energy from the wind farm, the grid price, along with the latest set of rolling forecasts of demands, grid prices and the energy from wind. If we have access to real data, we could use that. Otherwise, we would likely use Monte Carlo simulation which we described in Topic 2.

We need to write a simulator that we represent using our transition function

$$S_{t+1} = S^M(S_t,\ x_t, W_{t+1}),$$

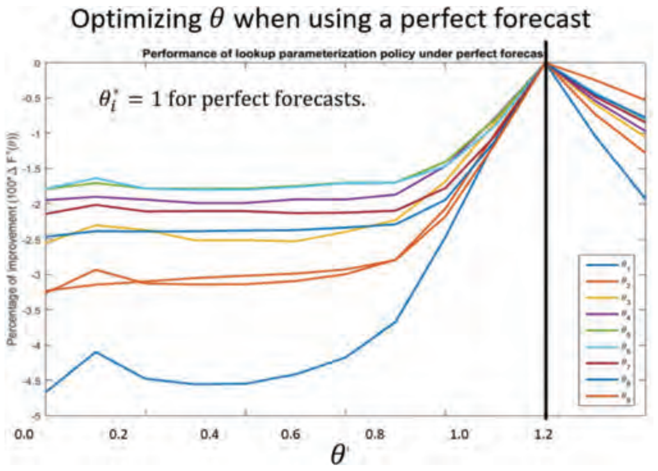
where S_t captures everything we know at time t, our decision comes from our policy $x_t = X_t^{D-LA}(S_t|\theta)$, and the new information W_{t+1} comes from our historical data or simulation. The transition function describes how our state variable changes over time (for example, this is where we update how much energy is in our storage device).

Running the simulation on the data W_t, with decisions from the policy $X_t^{D-LA}(S_t|\theta)$, allows us to estimate the performance of the policy:

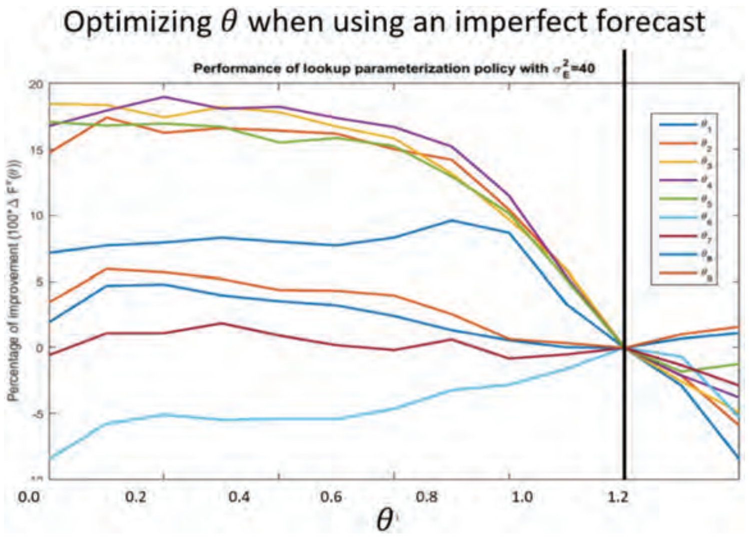
$$F^{D-LA}(\theta) = \sum_{t=0}^{T} C(S_t, x_t = X_t^{D-LA}(S_t|\theta)).$$

Next we have to turn to algorithms to search for the best value of θ. If we assume there is only a single parameter θ for all time periods in the future, this would just be a one-dimensional search. If it is a vector, we could use a gradient-based search such as what we illustrated in Topic 1 for nonlinear models. [See section 5.4 in RLSO for a more thorough discussion of how to compute gradients numerically.]

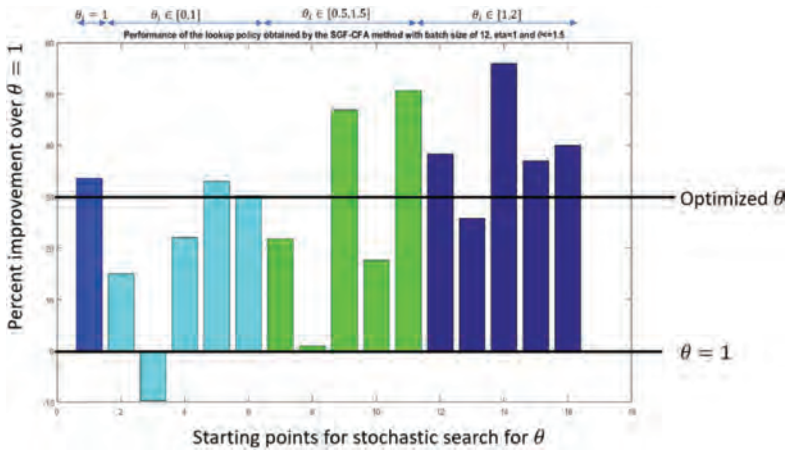
We can set up our simulation so that the rolling forecast is perfectly accurate. In this case, we would expect the best value of θ to be 1.0. The figure below confirms this.



Now run the experiment where the rolling forecasts (say of wind) is not accurate, as would occur in practice. In this case, we get the graph below, where the best values of θ are quite different from 1.0.



The figure below shows that we can achieve approximately a 30 percent improvement using optimized θ.



This application illustrates the very powerful idea of parameterizing an optimization problem. I suspect that there is a wide array of optimization problems that are actually policies for fully sequential problems. The idea of parameterizing an optimization problem is widely used in industry, but in an ad hoc way.

Topic 9: Integer Programming

We now introduce integer programming, using the classic problem of a facility location problem. Some schools teach entire courses on integer programming, so this lecture is more of a placeholder. An instructor can cover this material very briefly (e.g. at the level of these notes), or delve into the richer modeling challenges. We note that as of this writing, the best integer programming solvers (such as Gurobi or Cplex) are quite powerful for a wide range of problems.

As we did with linear programming, we are going to start with a basic integer programming problem which can be solved using commercial packages. It is for this reason that, as with linear programming, the traditional emphasis on algorithms is simply not appropriate for an introductory course.

We are going to introduce integer programming using a classical (and widely used) application of optimizing a set of locations to build or lease warehouses. This is a problem that can be solved to near optimality by high quality commercial packages. Unlike linear programs, integer programs come in different flavors, some of which are easier to solve while others still require specialized algorithms. In section 9.2 we provide a summary of some of the major classes of integer programming problems.

As we did with linear programming, we start with static models to illustrate integer programming. We follow in Topic 10 by extending the facility location problem to a dynamic setting.

9.1 Static Facility Location

A problem faced by many companies is locating facilities – these might be manufacturing points, distribution centers, warehouses, and even retail locations (if the company is a retailer).

Imagine that we are trying to design the network in the graphic below. We might assume we have a known manufacturing location in Mexico, but we have to optimize the location of distribution centers (black squares) and local warehouses (the smaller circles). To model this problem, let:

$I^{facility}$ = Set of candidate locations for distribution centers and warehouses (to be chosen).

I^{prod} = Set of production facilities (fixed in advance).

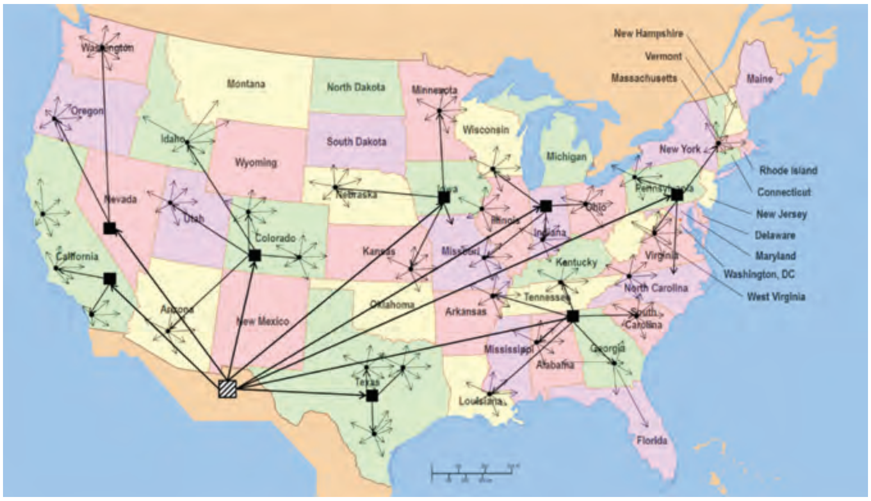
I^{retail} = Set of retailers where product is sold (fixed in advance).

x_{ij}^{trans} = Flow of goods (in pounds per quarter) from i to j.

$x^{trans} = (x_{ij}^{trans})_{i,j}$.

$$x_i^{facility} = \begin{cases} 1 & if\ we\ build\ facility\ i \\ 0 & Otherwise \end{cases}, i \in I^{facility}.$$

$x^{facility} = (x_i^{facility})_{i \in I^{facility}}$.



We can combine these into a single vector:

$$x = (x^{trans}, x^{facility}).$$

Next define the costs:

$c_i^{facility}$ = Cost per quarter to lease/operate facility i.

c_{ij}^{trans} = Transportation cost per pound for moving freight (in pounds) from i to j.

We have to move the freight from places where we have supplies, which includes initial inventories as well as the ability to produce the product (presumably only at the manufacturing facility). For this we let

q_i^{prod} = Production capacity of manufacturing facility $i \in I^{manuf}$, where we assume there is sufficient production capacity to meet the total market demand.

We then have to satisfy demands, given by

D_i = Demand for goods (in pounds) at retailer i over the quarter.

Our optimization problem to find the location of facilities can then be stated as

$$\min_{\mathrm{x}=(x^{trans},x^{facility})} \sum_{i \in I^{facility}} c_i^{facility} x_i^{facility} + \sum_{i,j \in I^{facility}} c_{ij}^{trans} x_{ij}^{trans} \tag{9.1}$$

This has to be optimized subject to the constraints:

$$\sum_{j \in I^{facility}} x_{ij}^{trans} \leq q_i^{prod} \qquad \text{for all } i \in I^{prod} \tag{9.2}$$

$$\sum_{k \in I^{facility}} x_{ki}^{trans} = D_i \qquad \text{for all } i \in I^{retail} \tag{9.3}$$

$$\sum_{i \in I^{prod}} x_{ij}^{trans} \leq q^{facility} x_j^{facility} \qquad \text{for all } j \in I^{facility} \tag{9.4}$$

$$\sum_{k \in I^{retail}} x_{jk}^{trans} \leq q^{facility} x_j^{facility} \qquad \text{for all } j \in I^{facility} \tag{9.5}$$

$$x_{ij}^{trans} \geq 0 \qquad \text{for all } i,j \in I^{prod}, I^{facility}, i^{retailer}, \tag{9.6}$$

$$x_i^{facility} \in (0,1) \qquad \text{for all } i,j \in I^{facility} \tag{9.7}$$

Equation (9.2) makes sure the total flow out of each production facility does not exceed the production capacity. Equation (9.3) requires that we meet the retail demand.

Equations (9.4) and (9.5) make sure that we do not ship out of or into any facility that has not been built, and if it has been built, that we do not exceed its capacity.

Equation (9.6) imposes the obvious condition that flows cannot be negative.

Equation (9.7) is where we insist that the facility variables $x_i^{facility}$ must be 0 or 1, and nothing in between. It is this constraint that makes this an integer programming problem.

Prior to year 2000, optimization problems with integer variables such as this could not be solved using commercial packages, and you can still find textbook authors referring to these problems as "hard." Today, commercial packages such as Gurobi (I think this is the leader) or its predecessor, Cplex, can handle a wide range of integer programming problems. The solution times will be slower than similar sized problems without integer variables, but the best packages can handle problems of realistic size without difficulty. Students do need to understand that while there are a number of freeware packages for solving linear programs, integer programs are harder, and the best packages may be quite a bit better than free software that you can get from the internet.

While we do make an attempt to indicate how linear programs can be solved with the simplex method (although I consider this optional – it really depends on the types of students), algorithms for integer programs are quite tedious. I strongly recommend leaving these to more advanced courses. Students learning optimization for the first time just have to understand that the complexity of integer programs means that more care has to be used when choosing a package.

9.2 Types of Integer Programs

It is important to recognize that there are major classes of integer programs, ranging from those that are no harder than solving a linear program to exceptionally hard problems that typically require specialized algorithms. Below is a list of some major classes of integer programs:

- Assignment problems (people/equipment to task) – These are problems where we are assigning discrete resources (people, machines) to discrete tasks. All the flows are 0 or 1. As long as a resource can only be assigned to at most one task, and each task requires only one resource, this is an easy problem that can be solved using a general purpose linear programming code and be guaranteed that the optimal decisions will be 0 or 1.

- Network flow problems as we illustrated in Topic 7 – This is another example of problems that can be solved using general purpose linear programming solvers, and still be guaranteed that the optimal solution will be integer as long as all the supplies, demands, and upper or lower bounds, are integer. Network flow problems are not limited to settings where a resource can cover just one task – the only limitation is that there can only be two types of constraints: flow conservation (flow out = flow in) and upper/lower bounds on flows.

- Network design (as we illustrated above) – Our facility design problem used to be considered a hard integer programming problem, but today the most advanced solvers (such as Gurobi or Cplex) can handle these problems. Run times will be much slower than if we drop the integrality constraints, but reasonable-sized problems (hundreds, even thousands of integer variables) can be solved with reasonable times.

- Vehicle routing problems – The simplest routing problem is the traveling salesman problem, and even this problem is beyond the capability of standard integer programming solvers. The problem arises when specifying constraints. It is easy to see that we need flow conservation constraints so that the flow into each node equals the flow out, but if we just include these constraints, it is possible to create cycles where a vehicle goes from city 1 to city 2 to city 3 back to city 1, without ever passing through the home depot for the vehicle. We can eliminate these "subtours" with "subtour elimination constraints" but we need an exponentially large number of these. Much harder problems include vehicles that have to make multiple stops to deliver goods. Even harder are problems where the vehicle must visit cities within a time window. There is an extensive literature on these problems.

- Sequencing and scheduling problems – These are arguably the hardest class of integer programming problems. These often arise when determining when to use a machine or trained technician to perform a set of jobs within time constraints (loose constraints are harder than tight ones). These problems tend to be solved using a class of methods known as constraint programming.

Topic 10: Dynamic Facility Location

The optimization problem we formulated in Topic 9 for our facility location problem is possibly one of the most standard problems used to illustrate integer programming. It is also widely used in industry, so we need to emphasize that this is a very useful model. However, it is important to understand how these decisions are actually made over time, and how the resulting network actually performs.

We begin by transitioning our original static, deterministic facility location problem into a two-stage problem where we first make the decision of locating facilities using forecasted demands to produce an estimate of the flows. Then, after we see the real demands, we reoptimize the flows. Our choice of where to locate facilities ignores the effect of these decisions on the future.

We then build on this "two-stage" model and use it as a policy for a fully sequential problem (in section 10.3) where decisions about which facilities to open or close are made sequential over time.

We start by describing the notation that we use.

10.1 Notation

To prepare for our dynamic models, we are going to begin by indexing all of our variables by t. So, our decision variables become:

$x_{ti}^{facility} = 1$ if we decide to activate facility $i \in I^{facility}$ at time t (we assume it becomes active right away). If it is already active, then $x_{ti}^{facility} = 1$ means to keep it active, while $x_{ti}^{facility} = 0$ means to deactivate it.

$\tilde{x}_{t,t+1,ij}^{trans}$ = the estimated flow of product from i to j in period $(t, t+1)$, using forecasted demands since this is all we know at time t. We are not going to implement these flows – these are estimates of what we think the flows *might* be using the information that does not become available until time $t+1$, using our best estimate (the forecast) at time t. We put a tilde on this variable so that it does not become confused with the decision variables that we are implementing (such as where to put a facility). We use a double time time index – the first index t indicates when we are making the decision which controls what we know, while the second time index captures the time period being modeled.

$\hat{D}_{t+1,i}$ = the demand at retailer i that does not become known until time $t+1$.

$f^{D}_{t,t+1,i}$ = the forecast of the demand $\hat{D}_{t+1,i}$ that is made at time t.

We need a variable for the flows that are made after we learn $\hat{D}_{t+1,i}$. We use:

$x^{trans}_{t+1,ij}$ = the actual flows that we determine after we see $\hat{D}_{t+1,i}$. Just as the facility decision $x^{facility}_{ti}$ is implemented using the information known at time t, $x^{trans}_{t+1,ij}$ would be the transportation flows that actually happen, given that they are computed using the actual demands $\hat{D}_{t+1,i}$.

We are next going to introduce variables for the *state* of facilities which we model using:

$$R^{facility}_{ti} = \begin{cases} 1 & if\ facility\ i\ is\ operating\ at\ time\ t \\ 0 & Otherwise \end{cases}$$

We can now make a decision to bring a facility into the network (if $R^{facility}_{ti} = 0$) or force it to leave the network (if $R^{facility}_{ti} = 1$). Our facility decision is then

$$x^{facility}_{t} = \begin{cases} 1 & If\ we\ want\ facility\ i\ in\ the\ network\ at\ time\ t+1 \\ 0 & If\ we\ want\ facility\ i\ out\ of\ the\ network\ at\ time\ t+1 \end{cases}$$

Of course, we can only add a facility if $R^{facility}_{ti} = 0$, and we can only drop a facility if $R^{facility}_{ti} = 1$.

We assume that if we add a facility at time t that it becomes available for the flows that are moved between t and $t+1$ while meeting the demands $\hat{D}_{t+1}$. We could introduce a longer delay, but it just complicates the model.

We have different costs for adding a facility to the network versus dropping it from the network, so we define:

$$c_{ti}^{add} = \text{Cost of adding facility } i \text{ at time } t.$$

$$c_{ti}^{drop} = \text{Cost of dropping facility } i \text{ at time } t.$$

We need variables that indicate whether we added or dropped facility i, or made no change, so we introduce

$$x_{ti}^{add} = 1 \text{ if we add a facility at } i, 0 \text{ otherwise.}$$

$$x_{ti}^{drop} = 1 \text{ if we drop a facility at } i, 0 \text{ otherwise.}$$

We then need to compute these variables using the "language" of linear constraints. We can do this with the following (remember that we want the smallest possible value of x_{ti}^{add} and x_{ti}^{drop}):

$$\begin{aligned} x_t^{add} &\geq x_{ti}^{facility} - R_{ti}^{facility}, \\ x_t^{add} &\geq 0, \\ x_t^{drop} &\geq R_{ti}^{facility} - x_{ti}^{facility}, \\ x_t^{drop} &\geq 0. \end{aligned}$$

We have to record the facility decision at time t in the facility state variable at time $t+1$:

$$R_{t+1}^{facility} = x_{ti}^{facility}.$$

10.2 Single-period Model with Uncertain Demands

In this section, we are going to argue that people often overlook the process of how a facility location problem is implemented. Typically it is understood that we solve the problem, and use the facility variables $x_i^{facility}$ to determine where we open or close facilities. On the other hand, we do not actually implement the flows contained in the flow variables x_{ij}^{flow} which are included in the model only as an approximation of what will actually happen in the field.

For example, the optimal solution might specify that a retailer gets their product from a particular warehouse (as indicated in the figure below), but on a particular day the warehouse may stock out, and the retailer would get their product from the next closest warehouse. We do not model these dynamics in the facility location model simply because it would make the model too large and complex. However, this means that our objective function (9.1) is nothing more than a rough approximation of how well the solution will perform in practice.

It is possible that the objective function (9.1) is a reasonable approximation, but not necessarily. A student used this model in a business game I was teaching at Princeton (famously known as the "orange juice game") to determine which of 50 possible locations should be used for warehouses. The cost of shipping to the warehouses was much lower than the cost of shipping from warehouses to retailers. As a result, the optimization model produced a solution recommending building a warehouse at each of the 50 locations.

The solution worked terribly in practice since it ignored the randomness in demands. When the solution was implemented, there were many stockouts because the model had not made any effort to capture the effects of random demands. When there are 50 warehouses, buffer stocks have to be larger in proportion to the averages. Using 5 warehouses means the flow through each warehouse is much larger, which produces a solution that is less sensitive to variations in the flow.

We are going to capture the effect of uncertainty by first assuming that we are going to locate our facilities using only forecasted demands.

Using our new notation, the constraints (9.2)–(9.7) in the static model become, at time t:

$$\sum_{j\in I^{facility}} x^{trans}_{tij} \leq q^{prod}_{ti} \qquad \text{for all } i \in I^{prod}, \tag{10.1}$$

$$\sum_{k\in I^{facility}} \tilde{x}^{trans}_{t,t+1,ki} = f^{D}_{t,t+1,i} \qquad \text{for all } i \in I^{retail}, \tag{10.2}$$

$$\sum_{i\in I^{prod}} \tilde{x}^{trans}_{t,t+1,ij} \leq q^{facility} R^{facility}_{t,j} \qquad \text{for all } j \in I^{facility}, \tag{10.3}$$

$$\sum_{k\in I^{retail}} \tilde{x}^{trans}_{t,t+1,jk} \leq q^{facility} R^{facility}_{t,j} \qquad \text{for all } j \in I^{facility}, \tag{10.4}$$

$$\tilde{x}^{trans}_{t,t+1,ij} \geq 0 \qquad \text{for all } i,j \in I^{facility}. \tag{10.5}$$

Finally, we include the allowed values of $x_{ti}^{facility}$: for $i \in I^{facility}$:

$$x_t^{facility} = \begin{cases} 1 & If\ we\ want\ facility\ i\ in\ the\ network\ at\ time\ t+1 \\ 0 & If\ we\ want\ facility\ i\ out\ of\ the\ network\ at\ time\ t+1 \end{cases} \tag{10.6}$$

The optimization problem for facilities is given by

$$min_{x_t^{facility}} \left(min_{\tilde{x}_{t,t+1}^{trans}} \sum_{i \in I^{facility}} c_i^{facility} x_{ti}^{facility} + \sum_{i,j \in I^{facility}} c_{ij}^{trans} \tilde{x}_{t,t+1,ij}^{trans} \right). \tag{10.7}$$

Solving (10.7) gives us the facility decisions $x_t^{facility}$ (which are implemented), and the planned transportation decisions $\tilde{x}_{t,t+1}^{trans}$ which are not implemented.

Once we make the decisions $x_t = (x_t^{facility}, \tilde{x}_{t,t+1}^{trans})$, we then have to evaluate the quality of the solution. The planned transportation decisions $\tilde{x}_{t,t+1}^{trans}$ were made based on forecasted demands $f_{t,t+1}^D$. However, we are going to assume that the actual transportation decisions are only made after we see the actual demands $\hat{D}_{t+1}$.

We first need to update the facility state variable $R_t^{facility}$ using the decisions $x_t^{facility}$ computed from solving (9.15) using

$$R_{t+1,i}^{facility} = R_{t,i}^{facility} + x_{ti}^{facility}. \tag{10.8}$$

To find the actual transportation decisions, we let

$x_{t+1,ij}^{trans}$ = the actual transportation flows based on the demands $\hat{D}_{t+1}$.

We do this by solving the problem above using the actual demands, and where the facility decisions have already been made (but not yet implemented). We can write this problem as

$$min_{x_{t+1}^{trans}} \sum_{i,j \in I^{facility}} c_{ij}^{trans} x_{t+1}^{trans} \tag{10.9}$$

which has to be solved subject to the constraints

$$\sum_{j \in I^{facility}} x^{trans}_{t+1,ij} \leq q^{prod}_{ti} \qquad \text{for all } i \in I^{prod}, \tag{10.10}$$

$$\sum_{k \in I^{facility}} x^{trans}_{t+1,ki} = \hat{D}_{t+1,i} \qquad \text{for all } i \in I^{retail}, \tag{10.11}$$

$$\sum_{i \in I^{prod}} x^{trans}_{t+1,ij} \leq q^{facility} R^{facility}_{t+1,j} \qquad \text{for all } j \in I^{facility}, \tag{10.12}$$

$$\sum_{k \in I^{retail}} x^{trans}_{t+1,jk} \leq q^{facility} R^{facility}_{t+1,j} \qquad \text{for all } j \in I^{facility}, \tag{10.13}$$

$$\hat{x}^{trans}_{t+1,ij} \geq 0 \qquad \text{for all } i,j \in I^{facility}. \tag{10.14}$$

Note that equation (10.11) is using the actual demands $\hat{D}_{t+1,i}$, whereas before we were using the forecasted demands $f^{D}_{t,t+1,i}$ in equation (9.9).

We now want to evaluate the solution $(x^{facility}_t, x^{trans}_{t+1})$. Our performance metrics can be divided between facility costs (including the cost of adding and dropping facilities), and the actual transportation costs. Facility costs are given by

$$C^{facility}(x^{facility}_t) = \sum_{i \in I^{facility}} (c^{facility}_i x^{facility}_{ti} + c^{add}_i x^{add}_{ti} + c^{drop}_i x^{drop}_{ti}), \tag{10.15}$$

$$C^{trans}(x^{trans}_{t+1}, \hat{D}_{t+1}) = \sum_{i,j \in I^{facility}} c^{trans}_{ij} x^{trans}_{t+1,ij}. \tag{10.16}$$

We assume that $x^{facility}_t$ is the optimal solution from solving (10.7), and x^{trans}_{t+1} is the optimal solution from solving (10.9)–(10.14). Of course, this means that we cannot compute $C^{trans}(x^{trans}_{t+1}, \hat{D}_{t+1})$ until after we have observed $\hat{D}_{t+1}$. Let the total costs be

$$C(x_t, \hat{D}_{t+1}) = C^{facility}(x^{facility}_t) + C^{trans}(x^{trans}_{t+1}, \hat{D}_{t+1}). \tag{10.17}$$

To evaluate our facility decision $x^{facility}_t$, we have to simulate different values of $\hat{D}_{t+1}$, and from this create different values of x^{trans}_{t+1}. Assume we create $n = 1, \ldots, N$ samples of the vector $\hat{D}_{t+1}$ which we designate

$$\hat{D}^1_{t+1}, \hat{D}^2_{t+1}, \ldots, \hat{D}^n_{t+1}, \ldots, \hat{D}^N_{t+1}.$$

For each sample $\hat{D}^n_{t+1}$, we compute a new set of transportation flows $x^{trans,n}_{t+1}$ which allows us to compute a new set of costs $C^{trans}(x^{trans,n}_{t+1}, \hat{D}^n_{t+1})$. Finally, we evaluate the cost of our facilities decision $x^{facility}_t$ using

$$\overline{C}^{facility}(x^{facility}_t) = C^{facility}(x^{facility}_t) + \frac{1}{N}\sum_{n=1}^{N} C^{trans}(x^{trans,n}_{t+1}, \hat{D}^n_{t+1}) \tag{10.18}$$

In the next section, we are going to recognize that we do not make facility decisions just once – these are made repeatedly over time. Since the decisions we make at time t depend on what facilities have already been created (and which have not), this means that our decisions now will impact the future, and need to understand how a decision now affects future decisions. In other words, it is a sequential decision problem!

10.3 Evaluating the Policy for a Multiperiod Problem

In the section above we evaluated the facility location policy for a single time period t, which illustrates the need to compute the transportation flows twice: first using forecasted demands, which we do in the optimization model where we choose the facilities, and then after we see the actual demands.

This realization highlights that we may not be doing as well as we could when we optimize facilities, since we are using point estimates of the demands that could produce a solution to the facility location problem that is vulnerable to variations in demands. We are going to address this issue in this section, but first we are going to transition to a full multiperiod setting, recognizing that we are not going to solve our facility location problem just once – we will need to keep solving it over and over.

In practice, it is likely that we would optimize flows daily, while we might reoptimize facilities monthly or quarterly. To keep the notation as simple as possible, we will continue to assume that we solve both problems at each time period.

In the previous section we averaged over N samples of $\hat{D}_{t+1}$. Here, we are going to simulate our policy over time, using just a single sample of $\hat{D}_{t+1}$ for time $t+1$. This time, however, we will simulate over a planning horizon $t = 0, 1, \ldots T$. We are going to introduce a minor notational change – we are now going to explicitly model *state variables* which capture the information we need when we make a decision. Since we have two decisions (and therefore two policies), we have two state variables: the facility state variable that captures the information we use to optimize facilities

$$S_t^{facility} = (R_t^{facility}, f_t^D),$$

and the transportation state variable which we use to optimize flows after the demands $\hat{D}_{t+1}$ have become known (at time $t+1$):

$$S_{t+1}^{trans} = (R_{t+1}^{facility}, \hat{D}_{t+1}).$$

We now write the optimization problem for determining the facilities (10.7) in the form of a policy

$$X_t^{facility}(S_t^{facility}) = \underset{x_t^{facility}}{argmin} \left(min_{\tilde{x}_{t,t+1}^{trans}} \sum_{i \in I^{facility}} c_i^{facility} x_{ti}^{facility} + \sum_{i,j \in I^{facility}} c_{ij}^{trans} \tilde{x}_{t,t+1,ij}^{trans} \right). \tag{10.19}$$

Note that we are only interested in $x_{ti}^{facility}$; we do not care about our determination of $\tilde{x}_{t,t+1,ij}^{trans}$ since we are only computing the transportation flows to help us find $x_{ti}^{facility}$.

After we optimize facilities, we update the vector that stores where we have facilities:

$$R_{t+1,i}^{facility} = R_{t,i}^{facility} + x_{ti}^{facility}. \tag{10.20}$$

After we determine $x_t^{facility}$, we assume we see the demands $\hat{D}_{t+1}$. This information, along with the decision $x_{ti}^{facility}$, determines the state for the transportation decision.

We next write the optimization problem for determining the flows (9.16) in the form of a policy

$$X_{t+1}^{trans}(S_{t+1}^{trans}) = argmin_{x_{t+1}^{trans}} \sum_{i,j \in I^{facility}} c_{ij}^{trans} x_{t+1}^{trans}. \tag{10.21}$$

Solving our transportation problem gives us x^{trans}_{t+1}. We then return to choosing the facilities $x^{facility}_{t+1}$ for time $t+1$, where we assume we are given a new set of forecasts $f^D_{t+1,t+2}$.

To evaluate our policies $X^{facility}_t(S^{facility}_t)$ and $X^{trans}_{t+1}(S^{trans}_{t+1})$, we simply extend what we did in the previous section to multiple periods. We can simulate a single sequence of demands $\hat{D}_1, \hat{D}_2, \ldots, \hat{D}_T$ (for this problem, the demands do not depend on decisions, so we can generate these in advance). Using these simulated demands, we can evaluate the performance of our policies using

$$\hat{F}^\pi = \sum_{t=0}^{T} (C^{facility}(X^{facility}_t(S^{facility}_t)) + C^{trans}(X^{trans}_{t+1}(S^{trans}_{t+1}), \hat{D}_{t+1})). \tag{10.22}$$

This approach evaluates the policy based on a single sample realization, which is just what we did in our machine learning problems in Topic 1, as well as the asset selling and inventory planning problems in Topic 2. We could create a sequence of samples

$$\hat{D}^n_1, \hat{D}^n_2, \ldots, \hat{D}^n_t, \ldots, \hat{D}^n_T, \quad \text{for } n = 1, \ldots, N.$$

We would then let $S^{facility,n}_t$ and $S^{trans,n}_{t+1}$ be the state variables created when following the n^{th} set of demands. We then simulate our policy N times and take an average:

$$\begin{aligned}\overline{F} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=0}^{T}&(C^{facility}(X^{facility}_t(S^{facility,n}_t))\\ &+ C^{trans}(X^{trans}_{t+1}(S^{trans,n}_{t+1}), \hat{D}^n_t)).\end{aligned} \tag{10.23}$$

At this point we can comment on the quality of the solution produced by our policies.

We note that the decision $x^{facility}_{ti}$ impacts both $S^{facility}_{t+1}$ as well as S^{trans}_{t+1}. If we assume that inventory is not held from one time period to the next, then it would mean that the transportation decisions x^{trans}_{t+1} do not impact the future, which means that our myopic policy (10.9)–(10.14) is optimal; otherwise, we can improve the policy by capturing the impact of decisions at time t on the future.

The facility policy $X_t^{facility}(S_t^{facility})$ in (10.19) is not optimal because facility decisions definitely impact the future. In fact, we anticipate that we could improve this policy significantly by capturing the impact of decisions on the future.

Our challenge now is designing better policies. However we design the policies, we can evaluate them using equation (10.23).

10.4 Alternative Facility Location Policies

There are two reasons why our policy for optimizing facilities is not optimal:

1. The transportation flows – Although we are not implementing the estimates of the flows $x_{t,t+1}^{trans}$, it is still the case that our facility decisions $x_t^{facility}$ depends on our approximate model of the transportation flows which depends on the forecasted demands rather than the actual demands.

2. Facility decisions made at time t, which depends on the facilities that are in use from the previous time period, $R_{t-1}^{facility}$. The facility decisions $x_t^{facility}$ then have an impact on $R_{t+1}^{facility}$, which would then have an impact on the facility decisions $x_{t+1}^{facility}$ in the next time period. Our policy does not capture the impact of decisions now on the future, so while we may be solving an optimization problem at time t, it would not be an optimal policy.

Below we introduce two new policies. The first is designed to handle the random demands as we did in section 10.2 for the single period model. The second policy is designed to handle limits on how many facilities we can add or drop in any time period. This constraint requires that we plan into the future. We illustrate this strategy using a simple deterministic lookahead model, paralleling how we solved the dynamic shortest path problem in section 5.2. Both of these policies would still need to be evaluated using the objective function in (10.23) which uses the average performance of the policy over N simulations.

10.4.1 Adjustment for random demands

We can modify our policy in a simple way to help with errors from using the forecasted demands $f^D_{t,t+1}$ instead of the actual demands $\hat{D}_{t+1}$. Using point estimates means we may not be prepared to handle sudden spikes in demand. Our constraints (10.12) and (10.13) require that we cannot exceed the capacity of the facilities, but while we might satisfy these constraints for the forecasted demands when we are making our facility decisions (as we did with equations (10.1)–(10.5)), we still have to satisfy the corresponding constraints (10.12) and (10.13) when we are using the actual demands.

A simple way to resolve this problem would be to introduce a buffer $\theta^{facility} < 1$ that factors down the capacity $q^{facility}$ within the facility policy, giving us modified versions of constraints (10.12) and (10.13):

$$\sum_{i \in I^{prod}} x^{trans}_{tij} \leq \theta^{facility} q^{facility} R^{facility}_{t,j} \qquad \text{for all } j \in I^{facility}, \tag{10.12a}$$

$$\sum_{k \in I^{retail}} x^{trans}_{tjk} \leq \theta^{facility} q^{facility} R^{facility}_{t,j} \qquad \text{for all } j \in I^{facility}. \tag{10.13a}$$

Our facility policy $X^{facility}_t(S^{facility}_t)$ (equation (10.19)) now depends on the parameter $\theta^{facility}$, which means we should write it as a parameterized policy $X^{facility}_t(S^{facility}_t|\theta^{facility})$. We would then write our objective function (10.22) as

$$\begin{aligned}\hat{F}(\theta^{facility}) = \sum_{t=0}^{T} (C^{facility}(S^{facility}_t, X^{facility}_t(S^{facility}_t|\theta^{facility})) \\ + C^{trans}(S^{trans}_{t+1}, X^{trans}_{t+1}(S^{trans}_{t+1}), \hat{D}_{t+1})).\end{aligned} \tag{10.24}$$

This gives us a new optimization problem, just as we have seen earlier (for example, in Topic 2) where we have to optimize the tunable parameter $\theta^{facility}$. We might write this as

$$\min_{\theta} \hat{F}(\theta^{facility}). \tag{10.25}$$

For this optimization problem to make sense we would have to introduce penalties for not satisfying demands in the event that total demand exceeds the capacities of the facilities.

The optimization problem in (10.25) is no different than any of the other parameter tuning problems we have seen for machine learning in Topic 1, or the parameter optimization problems for the policies in Topic 2.

There are other strategies we might introduce to capture the effect of changes to facilities at time t on future time periods, but the presentation here communicates the idea that a so-called optimal solution to an integer program does not mean that it is an optimal policy (and in fact it will virtually never be).

10.4.2 Deterministic lookahead model

A more realistic model of facility location would recognize that we are not going to open (or close) an entire set of facilities all at once. This is especially true if we actually have to construct the facility, but let's say we are just leasing space. However, let's say we have a limit on how many new facilities we can open or close each time period. We do this for simplicity; we might instead have a constraint on how much we can spend opening and closing facilities, but this simpler model will illustrate a way of planning into the future.

Recall from section 10.1 that $x^{add}_{ti} = 1$ if we open facility i, and $x^{drop}_{ti} = 1$ if we close facility i. We express the constraint on the number of openings and closing using

$$\sum_{i \in I^{facility}} (x^{add}_{ti} + x^{drop}_{ti}) \leq U^{facility}_t \tag{10.26}$$

Given the constraint (10.26), to make a decision of what to build now, we could optimize over a horizon $t, \ldots, t+H$. This is a lookahead model, so we use tilde's to indicate that these are variables for the lookahead model rather than the base model. Our decision variables are given by $\tilde{x}^{facility}_{t,t',i}$ and $\tilde{x}^{trans}_{t,t',i}$, where the index t captures that we are solving this problem at time t to determine the facility decisions $x^{facility}_{ti} = \tilde{x}^{facility}_{t,t,i}$. We optimize the decisions $\tilde{x}^{facility}_{t,t',i}$ for $t' = t+1, \ldots, t+H$ just to help us make the decision $\tilde{x}^{facility}_{t,t,i}$ that is actually implemented (as $x^{facility}_{ti}$). We still have to optimize the transportation decisions $\tilde{x}^{trans}_{t,t',i}$ but, as before, we have to model the transportation decisions to capture the interaction of facility location decisions on transportation costs.

We are going to solve this just as we solved the dynamic shortest path problems (section 5.2) where we optimize into the future, but only implement the decision in the first time period. We use a single set of forecasted demands made at time t, for all the time periods into the future:

$$X_t^{facility}(S_t^{facility}) = \underset{\tilde{x}_{t,t',}^{facility}, \tilde{x}_{t,t'}^{trans}, t'=t,\ldots,t+H}{\operatorname{argmin}} \sum_{t'=t}^{t+H} \left(\sum_{i \in I^{facility}} c_i^{facility} \tilde{x}_{t,t',i}^{facility} + \sum_{i,j \in I^{facility}} c_{ij}^{trans} \tilde{x}_{t,t',ij}^{trans} \right), \tag{10.27}$$

subject to the constraints for $t' = t, \ldots, t+H$:

$$\sum_{j \in I^{facility}} \tilde{x}_{t,t',jk}^{trans} \le q_{t',i}^{prod} \qquad \text{for all } i \in I^{prod}, \tag{10.28}$$

$$\sum_{k \in I^{facility}} \tilde{x}_{t,t',i}^{trans} = f_{t,t',i}^{D} \qquad \text{for all } i \in I^{retail}, \tag{10.29}$$

$$\sum_{i \in I^{prod}} \tilde{x}_{t,t',ij}^{trans} \le q^{facility} R_{t',j}^{facility} \qquad \text{for all } j \in I^{facility}, \tag{10.30}$$

$$\sum_{k \in I^{retail}} \tilde{x}_{t,t',jk}^{trans} \le q^{facility} R_{t',j}^{facility} \qquad \text{for all } j \in I^{facility}, \tag{10.31}$$

$$\tilde{x}_{t,t',ij}^{trans} \ge 0 \qquad \text{for all } i,j \in I^{facility}. \tag{10.32}$$

We then add a version of our constraint on the number of facilities to add or drop:

$$\sum_{i \in I^{facility}} (\tilde{x}_{t,t',i}^{add} + \tilde{x}_{t,t',i}^{drop}) \le U_{t'}^{facility} \quad \text{for all } i,j \in I^{facility}. \tag{10.33}$$

This problem is easy to write down, although solving it may be challenging even for commercial solvers. It was a major breakthrough when we could solve a single facility location problem which might be optimizing over 100 (or several hundred) possible locations for facilities. Now, we are multiplying that problem by the number of time periods in our planning horizon.

Assuming that we can solve the lookahead model, the policy $X_t^{facility}(S_t^{facility})$ needs to be simulated and evaluated using equation (10.23) just as we evaluated our original myopic policy. We can also introduce parameters as we did in section (10.2) to help accommodate the uncertainty in the flows.

Topic 11: Nonlinear Programming

As with linear and integer programming, we are going to start with classical “static” nonlinear programming problems, where we will introduce the formulation of portfolio optimization as a quadratic programming problem. We will then transition to solving this in a fully sequential manner.

11.1 Static Portfolio Optimization

Readings: This model is a streamlined version of the model in RLSO, section 13.2.4.

We are going to address a real problem solved by financial funds which requires that they continuously make decisions about how much to invest in a set of assets. The approach we describe here is based on an actual policy. It starts looking like a basic quadratic programming problem, but we are then going to see that it is really solved sequentially, and as a result it is a policy that needs to be tuned over multiple time periods as we have been doing in topics 7, 8 and 10.

We start by providing some notation:

$R_i =$ Amount currently invested in asset $i \in I^{asset}$.

$R_0 =$ Current cash on hand.

$p_i =$ Current price of asset i.

$p_i^{fcast} =$ Projected price in the future (say, 3 months out).

$c_i^{trans} =$ Transaction cost for buying or selling a share of asset i.

$x_i =$ Number of shares of asset i purchased (if $x_i > 0$) or sold ($x_i < 0$).

Our purchases and sales have to respect our available cash on hand, given by the constraint:

$$\sum_i p_i x_i \leq R_0. \tag{11.1}$$

The total transaction costs are given by

$$C^{trans}(x) = \sum_i c_i^{trans}|x_{ti}|, \tag{11.2}$$

where we take the absolute value since buying and selling transactions cost the same. To eliminate the absolute value (which complicates the formulation as an optimization problem), We first introduce a variable $x_i^{trans} = |x_i|$ that we compute by introducing two constraints:

$$x_i^{trans} \geq x_i, \tag{11.3}$$

$$x_i^{trans} \geq -x_i. \tag{11.4}$$

Using this new variable, our transaction costs can now be written

$$C^{trans}(x) = \sum_i c_i^{trans} x_i^{trans}. \tag{11.5}$$

A major issue in portfolio management is minimizing risk, which we measure by the standard deviation of the total return of the portfolio since the future price $\hat{p}_i$ of asset i will deviate from its current price p_i. We can use past data to compute the covariance matrix Σ of the prices of the stocks, where element Σ_{ij} is

$$\Sigma_{ij} = Cov(\hat{p}_i, \hat{p}_j). \tag{11.6}$$

The covariance $Cov(\hat{p}_i, \hat{p}_j)$ has units of "dollars squared." One way this information is sometimes presented is using the correlation coefficient ρ_{ij}. To compute this, we calculate the standard deviation of price $\hat{p}_i$ using

$$\sigma_i = \sqrt{Var(\hat{p}_i)}, \tag{11.7}$$

$$\rho_{ij} = \frac{Cov(\hat{p}_i, \hat{p}_j)}{\sigma_i \sigma_j}. \tag{11.8}$$

The correlation coefficient has the property that

$$-1 \leq \rho_{ij} \leq +1, \tag{11.9}$$

where $\rho_{ij} = 1$ means that the prices of assets i and j are perfectly correlated. Highly correlated stocks increase the volatility of the overall portfolio since if one stock drops, then other stocks that are highly correlated also crop, increasing the overall volatility of the portfolio.

We have two objectives when optimizing a portfolio. One is to maximize the return on the portfolio, where $p_i^{fcast} - p_i$ is our estimate of how much the price might be increasing (or decreasing). If we have R_i shares and purchase (or sell) x_i, we would then have $r_i + x_i$, and a measure of our total return would be

$$C^{return}(R,x) = \sum_i ((R_i + x_i)(p_i^{fcast} - p_i)) - C^{trans}(x). \tag{11.10}$$

The variance of the return of the portfolio is given by the quadratic formula

$$\begin{aligned} C^{risk}(R,x) &= \sum_i \sum_j R_i R_j Cov(p_i, p_j) \\ &= (R+x)^T \Sigma (R+x). \end{aligned} \tag{11.11}$$

We combine the return and risk in a single objective function that we write as

$$C^{total}(R,x|\theta^{risk}) = C^{return}(R,x) - \theta^{risk} C^{risk}(R,x). \tag{11.12}$$

Note that we subtract $C^{risk}(R,x)$ since this is something we wish to minimize. The parameter θ^{risk} handles the scaling problem since the units of $C^{return}(R,x)$ and $C^{risk}(R,x)$ are different.

Our optimization problem to determine the allocation x is given by

$$\max_x C^{total}(R,x|\theta^{risk}). \tag{11.13}$$

The optimization problem in (11.10) is a quadratic programming problem, which can be solved with several available packages. Of course, one algorithmic strategy is the methods we used in section 1.2 for fitting nonlinear machine learning models.

Setting the risk parameter θ^{risk} typically involves solving (11.13) for a range of values of θ^{risk} and then plotting $C^{return}(R,x)$ versus $C^{risk}(R,x)$ and then choosing the value of θ^{risk} that seems to strike the right balance for a particular situation.

Instructor: at this point go into as much detail as you want for nonlinear programming. Possible topics are:

- Gradient-based search, unconstrained and then constrained.
- Performing one-dimensional searches for gradient-based search
- Starting points
- Second-order algorithms.
- Mirror descent
- Nonconvex problems

11.2 Dynamic Portfolio Optimization

Readings: Section 13.2.4 from RLSO

The portfolio optimization problem in section 11.1 is clearly a problem that has to be solved repeatedly, over time, as new information is arriving. In other words, just as we illustrated with linear and integer programming, this is a sequential decision problem, where the optimization problem is actually a policy.

As with our facility location problem, we have to begin by indexing all the variables that are changing with a time index. This gives us the following notation:

R_{ti} = Amount currently invested in asset $i \in I^{asset}$ at time t.

R_{t0} = Current cash on hand at time t.

p_{ti} = Current price of asset i.

p_{ti}^{fcast} = Projected price in the future (say, 3 months out) given what we know at time t.

x_{ti} = Number of shares of asset i purchased (if $x_i > 0$) or sold ($x_i < 0$).

We are going to add an adjustment term that contains variables which we feel helps to reflect economic conditions. Examples of these variables might be

y_{ti1} = producer price index for asset i,

y_{ti2} = index of retailer inventories associated with asset i

y_{ti3} = manufacturing index for asset i.

As before, our transactions are limited by our budget constraint

$$\sum_i p_{ti} x_{ti} \leq R_{t0}. \tag{11.14}$$

For a dynamic system, we need to define the state variable S_t that captures all the relevant information at time t. For our problem, this would be

$$S_t = (R_t, p_t, p_t^{fcast}, y_t).$$

We still have our transaction variables

$$x_{ti}^{trans} \geq x_{ti}, \tag{11.15}$$

$$x_{ti}^{trans} \geq -x_{ti}, \tag{11.16}$$

which allows us to calculate our transaction costs using

$$C^{trans}(x_t) = \sum_i c_i^{trans} x_{ti}^{trans}. \tag{11.17}$$

Our portfolio return is still given by

$$C^{return}(S_t, x_t) = \sum_i \left((R_{ti} + x_{ti})(p_{ti}^{fcast} - p_{ti}) - C^{trans}(x_t)\right). \tag{11.18}$$

We are going to include an adjustment term using the economic variables y_t. We may want to modify each of these (square, log, ...), so we are going to write our adjustment term as

$$C^{adj}(y_t|\theta^{fcas}) = \sum_i \sum_k \theta_{ik}^{adj} \phi_k(y_{tik}). \tag{11.19}$$

We still have our risk component that we wish to minimize:

$$C^{risk}(S_t, x_t) = (R_t + x_t)^T \Sigma_{\mathrm{t}} (R_t + x_t), \tag{11.20}$$

where the covariance matrix Σ_{t} is computed from a rolling set of observations from time periods $t - H, \ldots, t$.

The total adjusted return that we wish to optimize is now

$$C^{total}(S_t, x_t|\theta^{risk}, \theta^{adj}) = C^{return}(S_t, x_t) + C^{adj}(y_t|\theta^{fcas}) \\ - \theta^{risk} C^{risk}(R_t, x_t). \tag{11.12}$$

Our policy is then to solve the following problem at time t:

$$X^\pi(S_t|\theta^{risk}, \theta^{adj}) = argmax_{x_t} C^{total}(R_t, x_t|\theta^{risk}, \theta^{adj}). \tag{11.13}$$

The way we can evaluate our policy is a process known in finance as "back testing" which is to take a historical sequence of prices for each asset i

$$p_{t-H,i}, \ldots, p_{t-1,i}, p_{ti},$$

along with a historical sequence of forecasts p_{ti}^{fcast}

$$p_{t-H,i}^{fcast}, \ldots p_{t-1,i}^{fcast}, p_{ti}^{fcast},$$

and the economic variables y_{tk}

$$y_{t-H,k}, \ldots, y_{t-1,k}, y_{tk}.$$

Using this data from history, we can evaluate our policy using

$$\hat{F}^\pi(\theta^{risk}, \theta^{adj}) = \sum_{t'=t-H}^{t} C^{total}(R_{t'}, x_{t'} = X^\pi(S_{t'}|\theta^{risk}, \theta^{adj})|\theta^{risk}, \theta^{adj}).$$

We assume that we have chosen a value for the risk parameter θ^{risk} as we described for the static model, but we can tune the parameters θ^{adj} using the optimization problem

$$\max_{\theta^{adj}} \hat{F}^\pi(\theta^{risk}, \theta^{adj}).$$

Once again we are tuning a policy that is itself a deterministic optimization problem, but this time it is a nonlinear programming problem.

Index